NAVAL POSTGRADUATE SCHOOL

MONTEREY, CALIFORNIA

THESIS

ELINT SIGNAL PROCESSING ON RECONFIGURABLE COMPUTERS FOR DETECTION AND CLASSIFICATION OF LPI EMITTERS

by

Dane A. Brown

June 2006

Thesis Advisor:	Douglas J. Fouts
Second Reader:	Herschel H. Loomis, Jr.

Approved for public release; distribution is unlimited

THIS PAGE INTENTIONALLY LEFT BLANK

REPORT DOCUMENTATION PAGE			Form Approved OMB No. 0704-0188
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instruction, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188) Washington DC 20503.			
1. AGENCY USE ONLY *(Leave blank)*	**2. REPORT DATE** June 2006	**3. REPORT TYPE AND DATES COVERED** Master's Thesis	
4. TITLE AND SUBTITLE: ELINT Signal Processing on Reconfigurable Computers for Detection and Classification of LPI Emitters			**5. FUNDING NUMBERS**
6. AUTHOR(S) Dane A. Brown			
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Center for Joint Services Electronic Warfare Naval Postgraduate School Monterey, CA 93943-5000			**8. PERFORMING ORGANIZATION REPORT NUMBER**
9. SPONSORING /MONITORING AGENCY NAME(S) AND ADDRESS(ES) National Security Agency Fort Meade, MD			**10. SPONSORING/MONITORING AGENCY REPORT NUMBER**
11. SUPPLEMENTARY NOTES The views expressed in this thesis are those of the author and do not reflect the official policy or position of the Department of Defense or the U.S. Government.			
12a. DISTRIBUTION / AVAILABILITY STATEMENT Approved for public release; distribution is unlimited			**12b. DISTRIBUTION CODE**

13. ABSTRACT (maximum 200 words)

This thesis describes the implementation of an ELINT algorithm for the detection and classification of Low Probability of Intercept (LPI) signals. The algorithm was coded in the C programming language and executed on a Field Programmable Gate Array based reconfigurable computer; the SRC-6 manufactured by SRC Computers, Inc. Specifically, this thesis focuses on the preprocessing stage of an LPI signal processing algorithm. This stage receives a detected signal that has been run through a Quadrature Mirror Filter Bank and outputs the preprocessed signal for classification by a neural network. A major value of this study comes from comparing the performance of the reconfigurable computer to that of supercomputers and embedded systems that are currently used to solve the signal processing needs of the United States Navy.

14. SUBJECT TERMS C Programming, Field Programmable Gate Array (FPGA), Hardware Description Language (HDL), Programmable Logic Device (PLD), Radar, Reconfigurable Computing, Signal Processing, Verilog HDL (VHDL)			**15. NUMBER OF PAGES** 105 **16. PRICE CODE**
17. SECURITY CLASSIFICATION OF REPORT Unclassified	**18. SECURITY CLASSIFICATION OF THIS PAGE** Unclassified	**19. SECURITY CLASSIFICATION OF ABSTRACT** Unclassified	**20. LIMITATION OF ABSTRACT** UL

NSN 7540-01-280-5500 Standard Form 298 (Rev. 2-89)
Prescribed by ANSI Std. 239-18

THIS PAGE INTENTIONALLY LEFT BLANK

Approved for public release; distribution is unlimited

ELINT SIGNAL PROCESSING ON RECONFIGURABLE COMPUTERS FOR DETECTION AND CLASSIFICATION OF LPI EMITTERS

Dane A. Brown
Ensign, United States Navy
B.S., United States Naval Academy, 2005

Submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE IN ELECTRICAL ENGINEERING

from the

NAVAL POSTGRADUATE SCHOOL
June 2006

Author: Dane A. Brown

Approved by: Douglas J. Fouts
Thesis Advisor

Herschel H. Loomis, Jr.
Second Reader

Jeffrey B. Knorr
Chairman, Department of Electrical and Computer Engineering

THIS PAGE INTENTIONALLY LEFT BLANK

ABSTRACT

This thesis describes the implementation of an ELINT algorithm for the detection and classification of Low Probability of Intercept (LPI) signals. The algorithm was coded in the C programming language and executed on a Field Programmable Gate Array based reconfigurable computer; the SRC-6 manufactured by SRC Computers, Inc. Specifically, this thesis focuses on the preprocessing stage of an LPI signal processing algorithm. This stage receives a detected signal that has been run through a Quadrature Mirror Filter Bank and outputs the preprocessed signal for classification by a neural network. A major value of this study comes from comparing the performance of the reconfigurable computer to that of supercomputers and embedded systems that are currently used to solve the signal processing needs of the United States Navy.

THIS PAGE INTENTIONALLY LEFT BLANK

TABLE OF CONTENTS

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF FIGURES

THIS PAGE INTENTIONALLY LEFT BLANK

LIST OF TABLES

THIS PAGE INTENTIONALLY LEFT BLANK

ACKNOWLEDGMENTS

I would like to thank David Caliga of SRC Computers, Inc. for his invaluable contributions to my research process. His hands on training course and book of SRC documentation made programming on the SRC-6 much simpler than it would have been otherwise.

Many thanks to Professor Jon Butler of NPS for his Advanced Computer Architecture course which gave me a first glance at the SRC-6 hardware and software interfaces.

I am grateful to Professor Phillip Pace of NPS who provided much of the research pertaining to LPI signal processing and answered many of my questions regarding that part of the project. His understudy, ENS Eric Zilberman, was also instrumental to my work as his research ran somewhat parallel to mine except that it dealt mainly in MATLAB.

Professor Douglas Fouts of NPS was my thesis advisor who organized and directed my research. He was also responsible for pointing me to sources of further help when I came to an impasse; I would like to thank him for his contributions.

Finally, this research was supported in part by the National Security Agency and the Office of Naval Research Code 313, Arlington, Virginia.

THIS PAGE INTENTIONALLY LEFT BLANK

EXECUTIVE SUMMARY

This thesis describes the implementation of an ELINT algorithm for the detection and classification of Low Probability of Intercept (LPI) signals. The algorithm was coded in the C programming language and executed on a Field Programmable Gate Array based reconfigurable computer; the SRC-6 manufactured by SRC Computers, Inc.

This thesis is part of a larger project in which all three stages of the detection to classification sequence for Low Probability of Intercept Signals are being ported to the SRC-6 and having their combined performance compared to general processing solutions that are in place presently. One student will be completing the porting of the detection algorithm into the C language for the SRC-6. The output from this code will then be fed into the section being completed in this thesis for preprocessing. The output from the code presented here will then be sent to the final stage of the detection to classification sequence which is classification of the modulations present in the signal by a trained Neural Classifying Network. A third student will be focusing his or her efforts on the creation and training of this Neural Network. The work from all three of these stages will be focused on the Quadrature Mirror Filtering Bank detection techniques, which is one of the common time-frequency techniques for detecting Low Probability of Intecept signals.

Specifically, this thesis focuses on the preprocessing stage of an LPI signal processing algorithm. This stage receives a detected signal that has been run through a Quadrature Mirror Filter Bank and outputs the preprocessed signal for classification by a neural network. Once these other two stages have been ported to the SRC-6 by the other students, all three stages may then be run on the SRC-6 hardware in parallel. These three pieces are optimized so that when they are run in parallel, the overall program should be much faster and provide a significant processing gain over general purpose processing solutions that are currently available. A major value of this study comes from comparing the performance of the reconfigurable computer to that of supercomputers and embedded systems that are currently used to solve the signal processing needs of the United States Navy.

THIS PAGE INTENTIONALLY LEFT BLANK

I. INTRODUCTION

A. BACKGROUND

In the field of battle, it is important to gain every possible advantage over an opponent. LPI radar and communication systems are one tool that helps to give this edge. LPI stands for Low Probability of intercept and these systems have characteristics that make them very difficult to detect with modern intercept receivers. Clearly, the edge could be taken back if one were able to come up with a reliable method to consistently detect and classify these signals in real time or near real time.

One approach seeks to determine if a reconfigurable computer can accomplish the required real-time processing of LPI radar and communications signals. ELINT (Electronic Intelligence) algorithms for automatically detecting and classifying LPI emitters have already been developed and tested. The specific ELINT algorithms applicable to Low Probability of Intercept emitters utilize the pseudo Wigner-Ville distribution algorithm, the Quadrature Mirror Filter Bank algorithm, and the Cyclostationary signal processing algorithm.

A reconfigurable computer is a computer with the capability to reprogram the hardware logic circuits and optimize them for the algorithms specific to the user [1]. These systems are usually based heavily on FPGAs (Field Programmable Gate Array) and PLDs (Programmable Logic Devices) embedded in their design. The advantage of reconfigurable computing is, as its name implies, that devices may be programmed and reprogrammed at any time to suit the needs of the user. This cuts down on the cost and development time of manufacturing an entirely new device simply to make minor changes to the hardware. Reconfigurable computers are also able to realize increased speed in their functions over general purpose supercomputers because the hardware is specialized and they are able to devote all of their processing power on the task they have been programmed to perform.

The SRC-6, by SRC Computers, Inc., is an example of such a reconfigurable computer. It contains two Pentium 4 processors and a Multi-Adaptive Processing (MAP) board with four Xilinx Virtex-II series XC2V6000 FPGAs. Through the provided Linux

interface, the SRC-6 can be programmed directly in a hardware description language that correlates specifically to the logic design inside the chip. More recently, SRC Computers has added the ability to port programs written in the C language over to the SRC-6. This greatly broadens the scope of what can be accomplished on the system.

Once running, the Virtex-II FPGAs operate at 100 MHz. This is clearly much slower than the speeds at which modern supercomputers operate, but the SRC-6 attempts to make up for its lack of speed in other areas such as its programmable hardware and its inherent parallelism.

B. OBJECTIVE

There are two major objectives of this thesis work. It is necessary to determine whether ELINT algorithms for the detection and classification of Low Probability of Intercept signals can be coded for execution on a reconfigurable computer. If it is indeed possible, the second goal will become to ascertain what level of performance can be achieved when compared to commodity computing solutions.

This research will help to answer questions regarding the practicality of reconfigurable computing, such as whether reconfigurable computers can process vast amounts of data in real time. Radar detection and classification can be a very time-sensitive matter in which information will be needed almost as soon as it is intercepted. This means that the real-time processing capabilities of the SRC-6 will be crucial in this study.

Once a suitable solution has been created and tested, its performance must be compared to the solutions that are already in place. Benchmarking will be necessary to compare the performance of the solution programmed onto the reconfigurable SRC-6 versus that of existing supercomputers and embedded signal processing systems. This benchmarking will compare these systems not only by quality of performance, but also by cost and speed. It is well known that supercomputers are very fast and that embedded systems are very specialized. It is anticipated that the SRC-6 reconfigurable computer will be able to obtain a desirable combination of these advantages at a much lower cost. If the SRC-6 is able to outperform the competition as expected, it could become an invaluable tool for the Navy to use in detecting and classifying LPI signals.

This thesis concentrates on the creation and examination of the preprocessing stage of the ELINT algorithms being coded onto the SRC-6. These algorithms are being coded in three distinct stages which are necessary to classify an LPI signal once it has been detected. The first of the stages is the detection itself which runs the intercepted signal through a filter created for LPI signals. This filtered signal is then passed on to a preprocessor which is covered in depth in this thesis. The preprocessor finally passes a resized image on to a neural network which performs the actual classification of the signal.

C. RELATED WORK

This thesis is part of a larger project which seeks to autonomously detect and classify LPI signals. The sequence of detecting and classifying these signals described above has been broken up into its major stages. The first and third stages which involve detection and classification, respectively, have each been assigned to another researcher for porting to the SRC-6. Thus, this thesis does not cover these areas in depth, rather it completely explains and creates code for the second stage, preprocessing, and details the porting of this code to the reconfigurable computer.

Some research in this field has previously been done by Professor Phillip Pace of the Naval Postgraduate School on using these algorithms on a general purpose computer. His work successfully simulated the detection to classification sequence using the software simulation package MATLAB. The MATLAB code he has developed is the basis for the MATLAB code which will be developed for the specific application of this thesis. This MATLAB code will be used as a model and a benchmark for similar code which will be created here for execution on the SRC-6.

D. THESIS ORGANIZATION

The remainder of this thesis is organized as follows:

- Chapter II provides an overview of the SRC-6 system. This begins with the system hardware, then moves on to a discussion of the software interface that has been provided.
- Chapter III is a discourse of the overall project, detailing each one of its major stages. These include the detection and filtering stage, the preprocessing stage, and the neural network classification stage.

- Chapter IV narrows in on the preprocessing stage and goes much deeper. This chapter describes the specific procedures of signal preprocessing which include cropping, thresholding, binarization, and matrix resizing.
- Chapter V describes the porting of the ELINT algorithms used from the original MATLAB code to the standard C code, and finally to the SRC-6 C code to be run on the reconfigurable computer.
- Chapter VI is a performance analysis of the final code running on the reconfigurable computer and a comparison of that implementation to other methods of detecting and classifying LPI radar signals.
- This thesis concludes with Chapter VII which gives a brief recap of the findings and makes some recommendations for future work in this area.

II. SRC-6 OVERVIEW

A. INTRODUCTION

The SRC-6 is a powerful reconfigurable computer made by SRC Computers, Inc. This chapter describes the reconfigurable computer and how it may help speed up signal processing applications. It begins with a discussion of the architecture of the SRC and the hardware interface that a user is provided. It then moves on to describe the software interface a user sees for programming the reconfigurable computer.

B. HARDWARE

As previously stated, the SRC-6 contains two Pentium 4 processors and a Multi-Adaptive Processing (MAP) board with four Xilinx Virtex-II series XC2V6000 FPGAs.

1. Microprocessor

The Intel microprocessor is separate from the MAP board and this is where the general purpose computing takes place, just as in an ordinary computer. It also has a memory bus and access to the system Common Memory which can be accessed by both the microprocessor and the MAP. The SRC provides a SNAP™ card to interface between the microprocessor and the MAP via Direct Memory Access procedures [2].

2. MAP Board

a. MAP Organization

The MAP processor consists of some general control logic, memory, and two Xilinx Virtex II series XC2V6000 FPGAs. There are two MAP processors on the MAP board. Figure 1 below gives a block diagram of the MAP. It contains input and output to System Common Memory (SCM) to receive commands. It also has the MAP Control Processor which contains the system flags and the data registers. The MAP provides hardware for User Logic to be configured by the end user. In addition, there is an On-Board Memory bank that interfaces with the User Logic and the Control Logic via 62-bit data ports. The MAP board has General Purpose Input Output (GPIO) ports which allow direct connections for other MAPs or data input. It also contains Direct Memory Access (DMA) engines which support: distributed SRAM, Block SRAM, On-Board SRAM, and microprocessor memory [3].

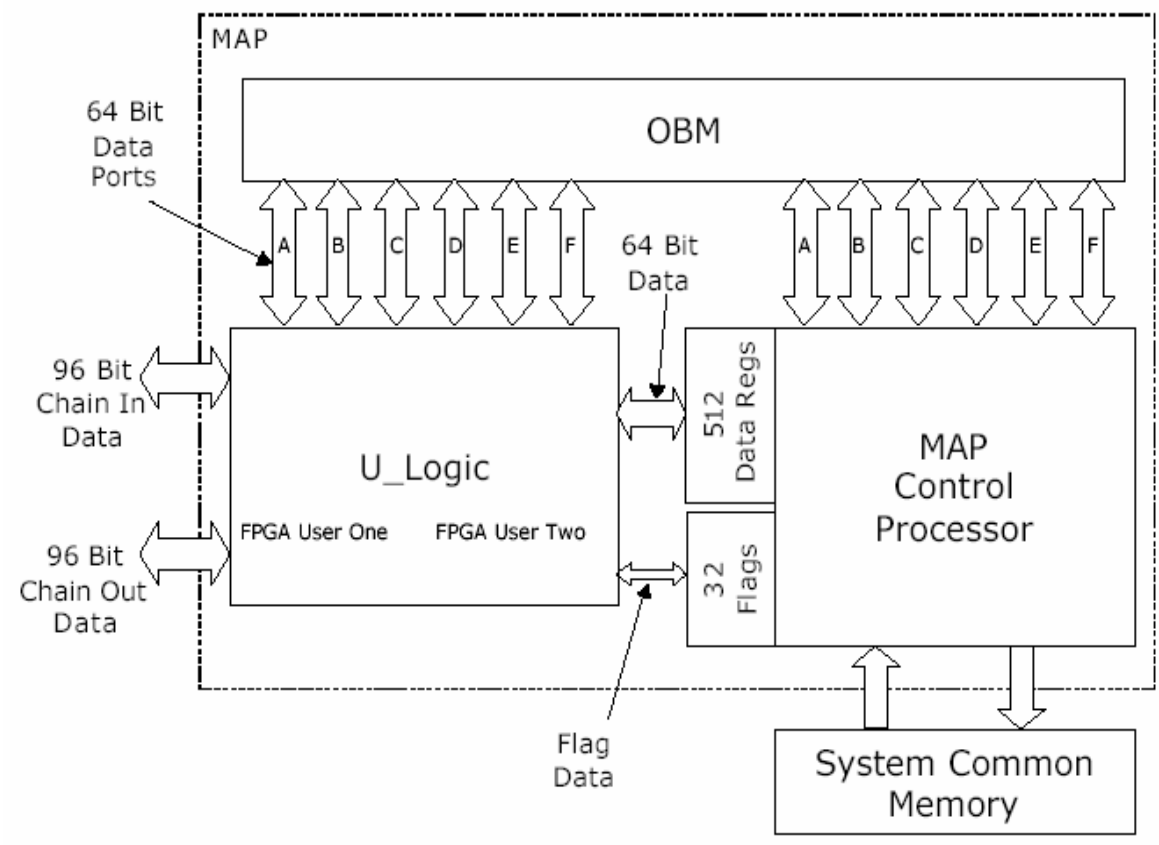

Figure 1. Interface Architecture of the MAP (From Ref. 4.)

b. Control Logic

MAP operation is coordinated through use of Control Logic in the MAP Control Processor. This processor receives instruction sequences from a Command List, or ComList, which is provided by the System Common Memory via a DMA engine. This Control Logic executes instructions sequentially and controls such tasks as: User Logic functions, operand input, and result output [4].

The Control Processor is where the MAPs 32 flags and 512 data registers are found. Table 1 provides an appropriate description of these resources.

Table 1. MAP Registers (From Ref. 4.)

Registers	Description
Data Registers	Data Registers (DRs) are mainly used to hold addresses, (both SCM and OBM addresses), but can hold any needed data. By using Data Registers, scalar (single item) data can be sent to or received from User Logic. Only simple arithmetic and logical operations are supported on these registers. Register contents can be tested for zero/non-zero and thus support command branches and loops. A given ComList command must use DRs from within the same group. There are sixteen groups of thirty-two 64-bit registers, known as Groups 0 through 15 of DR0 through DR31. DR0 is always zero. DR1 is always an integer 1. This applies to all groups. Neither register can be written, and errors will not be reported if a write to DR0 or DR1 is attempted.
Flag Registers	Flag Registers (FRs) are used for direction of movement, requests, and complete functions. Commands can test and wait on the state of Flag Registers before execution (e.g. hold execution of this ComList command until Flag 3 goes set). Commands can change the state of Flag Registers, forcing them to set and clear. In addition there are logical instructions that allow combining Flag states (AND, OR, XOR). Other commands allow testing and branching on the contents of Flag Registers. There are thirty-two single-bit Flag Registers, FR0 through FR31 FR0 is always clear, 0, and FR1 is always set, 1. Neither register can be written, and errors will not be reported if a write to FR0 or FR1 is attempted. FR31 is reserved for future use.
Temp Register	A register in the Control Processor used for temporary storage during the TMP2DR and DR2TMP commands.

The explicitly controlled logic circuitry also features what is known as Direct Execution Logic (DEL). Direct Execution Logic is comprised of at least one User Logic device. These circuits allow for explicit computational units, memory pre-fetch units, and data access units [3]. These features cause the SRC-6 to have more efficient use of logic gates, power, and bandwidth, thus increasing processing power by several orders of magnitude, as compared to existing solutions.

c. Memory

The SRC-6 MAP has access to its own memory known as On-Board Memory (OBM). There are six banks with each having four megabytes of memory, for a total of 24 megabytes. These banks are essentially arrays where each data element is exactly 64-bits long; each array can hold up to 523,776 elements, for a total of 3,142,656

64-bit data words across On-Board Memory. Technically, each bank could hold as many as 524,288 data elements but 512 of these data words are reserved for scalar values and working space.

The Direct Memory Access (DMA) engines are what allow this On-Board Memory system to be so effective. Data is transferred between the OBM banks and the general purpose microprocessor via DMA transfers. When transferring data to or from On-Board Memory banks one must specify: the direction of the transfer, where the OBM banks begin, how often to stripe, or input, the data, the Central Processing Unit (CPU) address, the CPU stride, the total length of the transfer in byte, and finally the server number which must wait for the DMA transfer to complete [5]. The instruction format is as follows:

```
DMA_CPU (<Transfer Direction>, <OBM address>, <OBM striping>, <Computer Memory Address>, <Computer Memory Stride>, <Length>, <Server>);
wait_DMA (<Server>);
```

The following, Figure 2, shows a clear illustration of how each On-Board Memory bank interfaces with the User Logic. Data is sent back and forth in the standard 64-bit word format that OBM banks accept. Since there are 512k words in the memory banks, User Logic uses a 19-bit address bus to decode which element is being accessed. Finally, to control the bidirectional data, there is an FPGA output enable bit, a write enable bit, and a read enable bit.

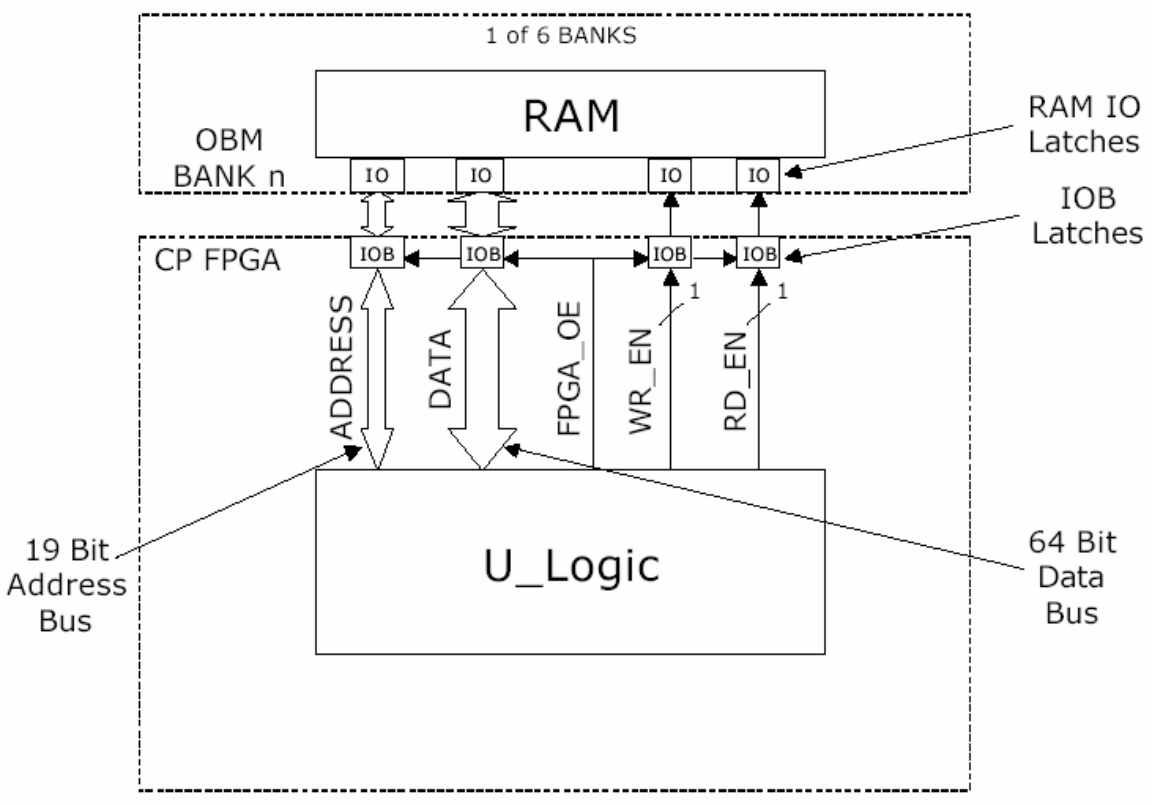

Figure 2. On-Board Memory Interface (From Ref. 4.)

Having these six banks truly enhances the parallelism of the system, which is one of its greatest strengths, because each one can be accessed simultaneously.

d. FPGA

The User Logic recently discussed is comprised of two Xilinx Virtex II Field Programmable Gate Arrays (FPGAs). These FPGAs are what make the SRC-6 a reconfigurable computer; the end users program them to their own specifications and thus define what is contained in User Logic. The User Logic FPGAs can interact with the control circuitry and can also read and write to On-Board Memory through various ports.

This diagram delineates the interfacing of the FPGAs to the rest of the MAP. It shows the six 64-bit bidirectional ports to On-Board Memory, as well as the ports to the MAP Control Processor through the flag bit and the data register bits. It also illustrates the input and output chain ports that could connect the local User Logic to external User Logic on another MAP, thus forming a chain of MAPs.

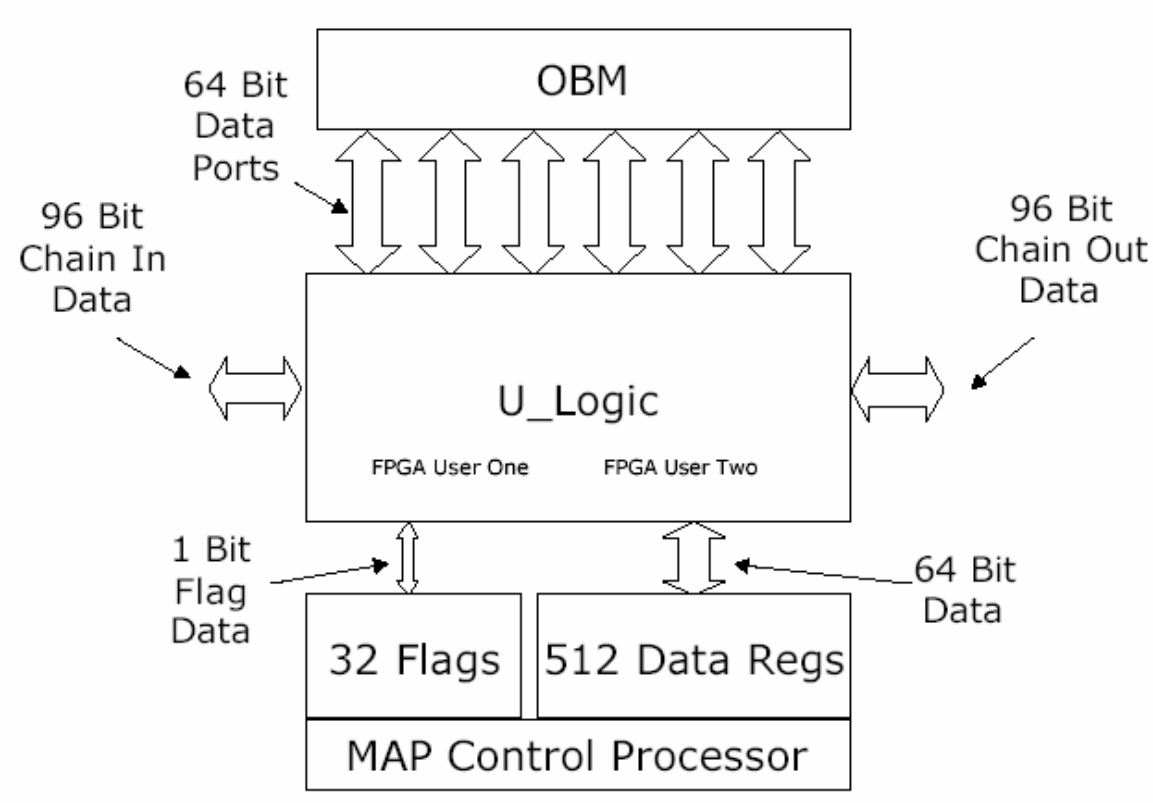

Figure 3. User Logic Interface to MAP (From Ref. 4.)

Since MAPs can be chained together by User Logic, they can communicate with one another, send partial results to other MAPs, and receive partial results from other MAPs. In this way, another dimension of parallelism is added to the SRC-6, which can reap great benefits in performance.

Not only can the FPGAs communicate with the rest of the MAP and other MAPs, they can also communicate between each other. The two FPGAs in User Logic are able to send data to one another using a 64-bit data bus that is separate from the memory board and does not use up any memory bandwidth. This bus, when combined with three bits of user defined input and three bits of user defined output, constructs a Bridge Data Port. There are three Bridge Data Ports [4].

Figure 4 which follows shows how the FPGAs communicate.

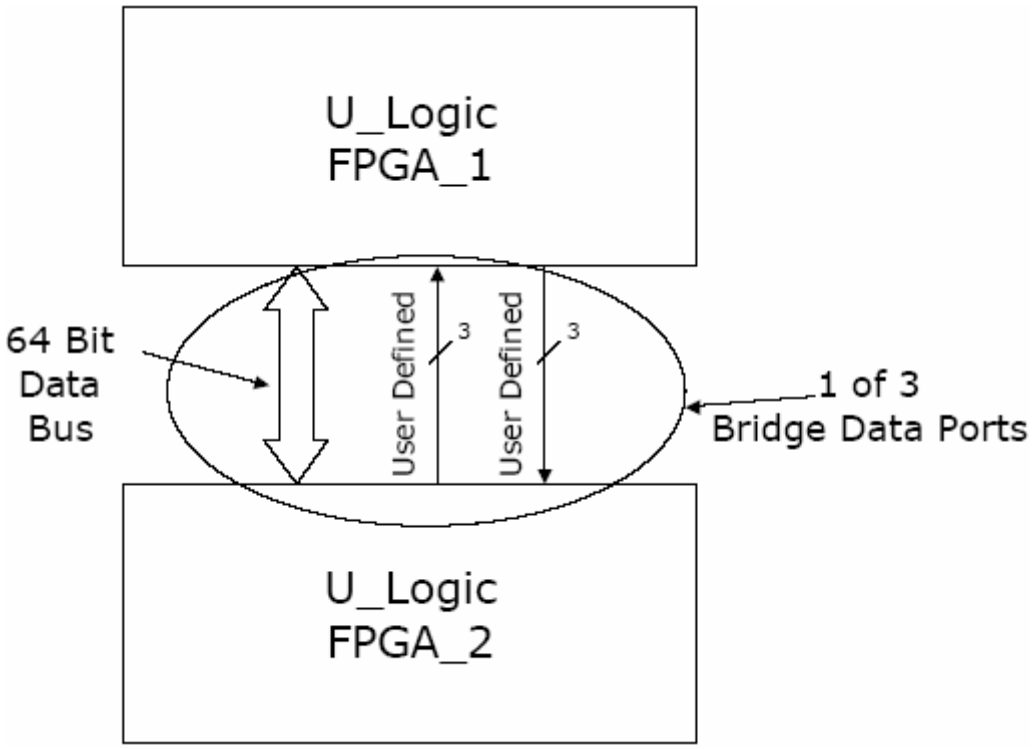

Figure 4. User Logic Internal Interface (From Ref. 4.)

C. SOFTWARE

1. Software Environment

In order for the end users to utilize the functionality of the reconfigurable computer, they must program the SRC-6 using a specialized software development environment on a Linux computer.

a. Linux Operating System

Currently, the SRC-6 is compatible with the Red Hat version of the Linux operating system (OS). Linux is a Unix-based operating system which is not as familiar to most people or as user-friendly as Microsoft Windows, however it has many tools available for experienced users that really ease the processes of writing code, testing it, and debugging it.

Another advantage to using Linux is the shell interface. Unlike in Windows, one can easily set up a remote shell to a computer to which access is granted and execute any command as if one were at the host computer. This eliminates the need for a programmer to be physically located at the SRC-6 in order to program it. This

ability is especially convenient at facilities where there is only one SRC-6 because multiple users can log onto it remotely and test their code rather than having to wait at the one machine for each person to complete their work.

b. Languages

As with any computer program, the creator must write and compile the code that accomplishes the desired function. For most reconfigurable computers, the language in which this code is written is VHDL, or VHSIC Hardware Description Language, which uses practical mnemonics to describe the specific hardware functions. While useful, VHDL still has a steep learning curve. It is still a relatively low-level language with which many programmers are unfamiliar.

SRC Computers, Inc. has attempted to eliminate the need for writing VHDL code, thus relieving the burden on the code developer. When programming the SRC-6, the coder has a choice between programming in FORTRAN or C. Both of these are very high level languages that do not possess the same control over the hardware that VHDL does, but each one is more intuitive to experienced programmers. Though FORTRAN is more straightforward than VHDL, the language itself is antiquated and obsolete. However, it is still an option to a programmer who is comfortable with it [5].

The overwhelming choice for modern programmers is C. While it does not include the object oriented abilities of its successor C++, it is equipped to handle most tasks applicable to reconfigurable computing and it is used exclusively in this thesis.

c. File Types

There are two file types made available to the programmer who codes in C, they are .c files and .mc files. These extensions tell the SRC-6 where to run the code. The .c files are set to be executed on the microprocessor, whereas the .mc files are set for execution on the MAP. Both files generally include *libmap.h*, a. header file that defines the prototypes and constants specific to working with the SRC-6. This allows the use of SRC-6 specific data types such as *uint64_t* and functions such as DMA transfers.

To optimize programs on the SRC-6, a user must decide ahead of time what should be run on the microprocessor and what should be run on the MAP. The FPGAs can be configured to run certain repeated calculations much faster and more

efficiently than the general purpose microprocessor. Once the programmer has planned the optimal method, the code that is desired to run on the MAP is written into the .mc file while the code that must be executed on the general purpose microprocessor is written into the .c file. Usually, the .mc file is used to run optimized functions and the .c file will be used to pass in the necessary parameters to be manipulated and then receive the desired data.

As previously noted, the C programs do not give the coder much specific control over hardware functions. If a user does need more hardware control, macros can be written directly in VHDL. There are two types of useful user-defined macros, external macros and user defined macros. External macros interact with the outside SRC-6 system and are controlled by *start* and *done* signals received from the system. Purely functional macros receive inputs and compute outputs every clock cycle. They are not stateful and can only process current data that they are given. However, they are able to pipeline data and execute them in parallel.

Macros that are created have three parts. The first is the VHDL code that describes the functioning of the hardware; this can be explicitly written by the programmer or extracted from a graphic schematic editing tool. The second part is a black box file, this file depicts the input and output interfaces of the macro. Finally a macro needs an info file. Info files work by linking the hardware signals to their respective variable and functions that they will be called with from the C programs [5].

d. Editing Programs

Editing C programs in Linux is a bit different than working with them using Microsoft Windows. The standard editors are not as user friendly and do not walk the programmer through the entire process of editing, formatting, compiling, and testing. *Vi* is a basic text editor found in all distributions of Unix and Linux. It is difficult to learn how to use, but if a programmer understands it, it can be used to program on any Unix-based machine. Another standard tool that is generally found is *gedit.* This is a graphical text editor that may be slightly more familiar to the average user, but still not very helpful to someone specifically trying to write programs. A rather helpful editor that is found on

some Linux platforms is *emacs*. This is another graphical tool that is a powerful program editor. It recognizes languages like C and automatically formats and color-codes text to make it simpler for a coder to read.

2. Compilation

Just as in any other programming language, written code must be compiled in order to be tested and debugged. In general, when programming in C on a Linux platform the GNU C compiler, *gcc*, is used. This program compiles the source code into object code then links and assembles it and turns it into an executable program. This is the general process for turning a high-level program into a running executable. However, it is not acceptable for SRC-6 programming. The C code must be translated into VHDL so that the FPGAs can understand how to operate the hardware.

This calls for a different compilation method which involves a Make file. The Make file is the last required file in a project directory. It goes alongside the .c, the .mc, and the macro files and is always titled *Make file*. There are three ways in which a user may Make a project. The first is a *make debug* mode. Debug compilation is fast because the code is just interpreted; if the syntax in the files is correct, debug mode will quickly simulate the results of the executed code. If the programmer wants to see a simulation that more closely emulates what will happen on the MAP, a *make simulate* mode is available. A simulated program takes longer to Make, but not as long as a full compilation. The final debug mode available to the coder is the *make hardware* mode. This actually Makes the code in the SRC-6 hardware ready for execution. This mode, however, takes much longer than the others to complete and is not practical for testing and debugging code. It is best used after completion of the debugging stage for performance analysis.

Figure 5 below is an accurate visual aid in understanding the stages of the compilation process that take place on the SRC-6.

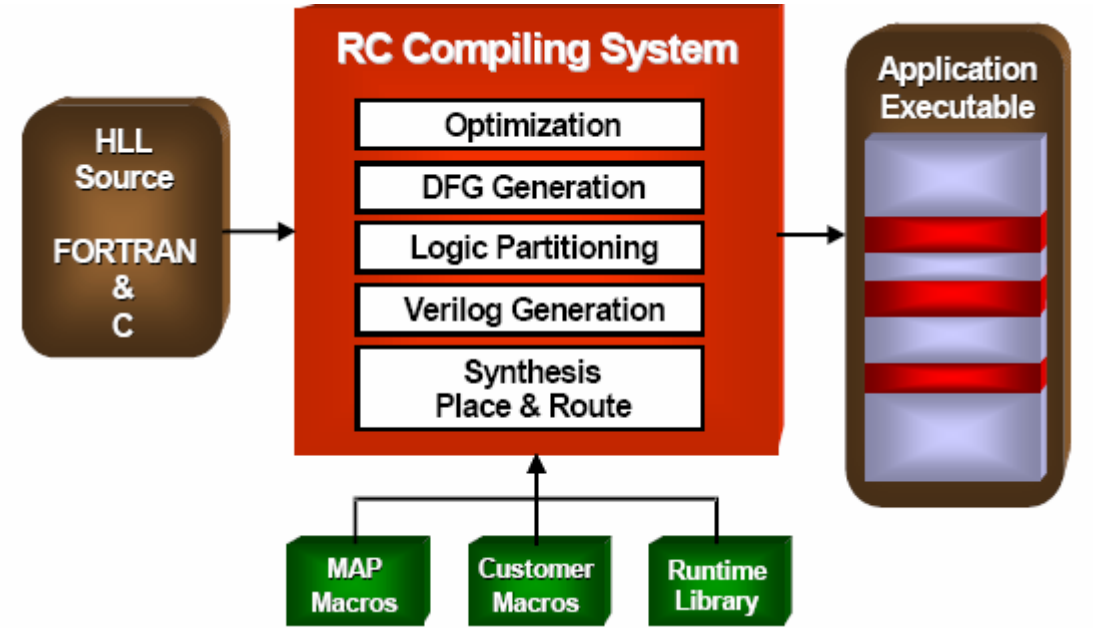

Figure 5. SRC-6 Compilation Process (From Ref. 5.)

The previous *make* options compile the project in a certain way and leave many files in the project directory specific to their respective modes of execution. In order to recompile the program, a user must restore the directory to the basic source files. This is accomplished with the *make clobber* command. Another option is the *make clean* command, this will perform a similar function to *make clobber* with the exception that it will also leave the executable files themselves in the project directory [5].

D. SUMMARY

This chapter provided a brief overview of the hardware and software interfaces provided in the SRC-6. It began with a description of the major hardware components, to include the microprocessor, the Multi-Adaptive Processing Board, and their interfaces. It then went on to discuss the software environment presented when working with the SRC-6, as well as how the compilation process works.

This information is necessary to the research because in order to program the reconfigurable computer, one must have a solid understanding of its interfaces and how to use them to accomplish the necessary tasks. The upcoming chapters move into a discussion of the process of detecting and classifying Low Probability of Intercept radar.

THIS PAGE INTENTIONALLY LEFT BLANK

III. PROJECT STAGES

A. INTRODUCTION

Low Probability of Intercept signals are difficult to detect because of their inherent properties. For example, LPI radar has very low power, a wide bandwidth, and tends to vary the transmission frequencies. Additionally, LPI signals have the advantage of being veiled by modern environmental effects to include high noise interference and multiple signals destructively interfering with one another [6]. Clearly, an edge could be taken if one were able to come up with a reliable method to consistently detect and classify these signals in real time.

Non-cooperative intercept receivers achieve a significant increase in processing gain when using time-frequency and bifrequency techniques to detect LPI radar modulations. It is possible to use parallel pattern classification with these detection techniques to autonomously determine the modulation present in a signal. This intelligent autonomous handling of the intercepted signal will greatly mitigate if not eliminate the need for human intervention which can ultimately lead to real-time handling of the data [6].

There are three stages involved with intercepting an LPI signal and determining what it is. These are detection, preprocessing, and classification. If these three stages can be programmed into a pipeline and run in parallel on an SRC-6, near real-time processing of the LPI signals can be achieved.

The flow of these three stages can be seen in Figure 6 to follow. This image is a good general description of the project and will be a constant visual reference for this thesis.

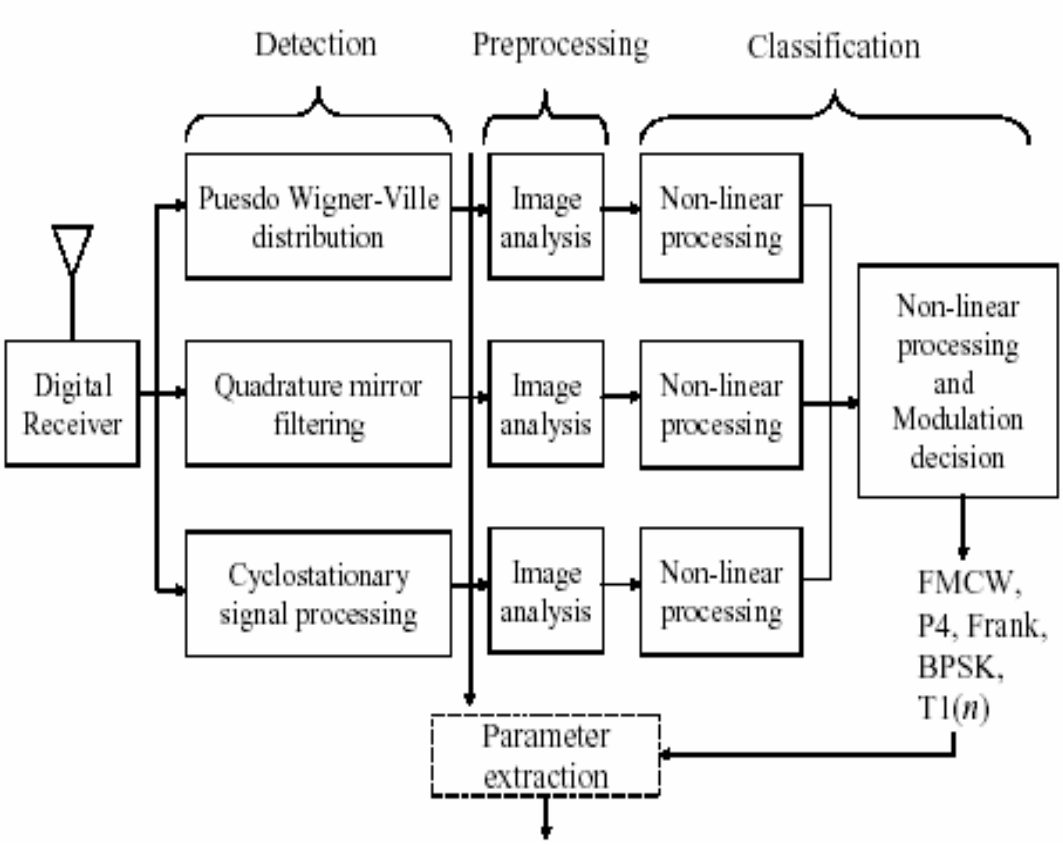

Figure 6. Project Flow Chart (After Ref. 6.)

B. DETECTION

The detection can be performed by any one of three algorithms. The first is a pseudo Wigner-Ville distribution. The second is by a Quadrature Mirror Filter Bank (QMFB). The third method of detection is through Cyclostationary signal processing. The pseudo Wigner-Ville distribution and the Quadrature Mirror Filter Bank are both time-frequency techniques. Cyclostationary signal processing is a bifrequency detection technique. For simplicity, the entire process may be viewed as it stems from just one detection technique. A reasonable choice is to focus on the Quadrature Mirror Filter Bank.

1. Detection Techniques

Time-frequency and bifrequency techniques are advanced signal processing methods that are used to detect Low Probability of Intercept signal because the wide bands of their frequency and phase modulations require significant processing gain on the part of the receiver. These two techniques produce images that can be later classified by a trained human or an autonomous device.

a. Time-Frequency

Time-frequency distributions are generally used in examining non-stationary signals. These distributions represent the frequency spectrum of signals as a function of time, which make it relatively simple to determine what kind of modulations are present in a given signal. An ordinary time-domain signal lacks details about the frequencies of a signal making it impossible to determine the frequency modulations. Taking a Fourier transform of a time signal provides its frequency spectrum, however this spectrum does not specify anything about the times at which the signal energies are present.

Time-frequency distribution is a more sophisticated technique which takes advantage of the fact that time information is encoded into the phase of a Fourier transform. Taking this time information from the frequency spectrum is rather difficult as it involves complex calculations like phase unwrapping. Time-frequency information is usually a direct representation of the frequency content of a signal while keeping the time parameter intact [6]. These time-frequency techniques apply themselves well to detection techniques such as the Wigner-Ville Distribution and the Quadrature Mirror Filter Bank Tree.

b. Bifrequency

Bifrequency spectral analysis is a method of taking a periodic part of a signal and examining its spectral frequency versus its cycle frequency. A bifrequency map describes a Linear Time Varying (LTV) system. For a non-stationary vector random process its autocorrelation function is a function of two indices. These processes have a bispectrum matrix which is a two-dimensional Fourier transform. This matrix fully describes the second order statistics of the process [7].

2. ELINT Algorithms

ELINT algorithms are Electronic Intelligence algorithms. The specific ELINT algorithms applicable to Low Probability of Intercept signal processing utilize the pseudo Wigner-Ville distribution algorithm, the Quadrature Mirror Filter Bank algorithm, and the Cyclostationary signal processing algorithm.

a. Wigner-Ville Distribution

The first type of time-frequency algorithm used in detecting LPI signals is the Pseudo Wigner-Ville distribution (PWVD). This is one of the most useful and popular methods of time-frequency analysis in signal processing. The output of this distribution is always real and produces cross-terms between each pair of signal components which can complicate the recognition of signal modulations. These cross-terms can, however, help the classification process with the added information they provide [6].

b. Quadrature Mirror Filter Bank Tree

Another ELINT algorithm which makes use of time-frequency analysis is the Quadrature Mirror Filter Bank (QMFB) Tree. For the purpose of simplicity, this project follows the detection, preprocessing, and classification procedures using only one of the detection techniques in the above flow chart; the chosen technique here was the Quadrature Mirror Filter Bank Tree.

A QMFB tree consists of a number of layers of fully connected pairs of orthogonal wavelet filters (or basis functions) that linearly decompose the received waveform into tiles on the time-frequency plane. A modified sinc filter is used and every filter output is connected to a filter pair in the next layer, as shown in Figure 7 below. The tiles are used to refer to the rectangular regions of the time-frequency plane containing the basis function's energy. Each filter pair divides the digital input waveform into its high-frequency and low-frequency components, with a transition centered at π. Within the series of time-frequency layers, each subsequent layer provides a trade-off in time and frequency resolution. By examining the energy within the tiles, parameters such as bandwidth, center frequency, phase modulation, signal duration and location in the time-frequency plane can be determined [6].

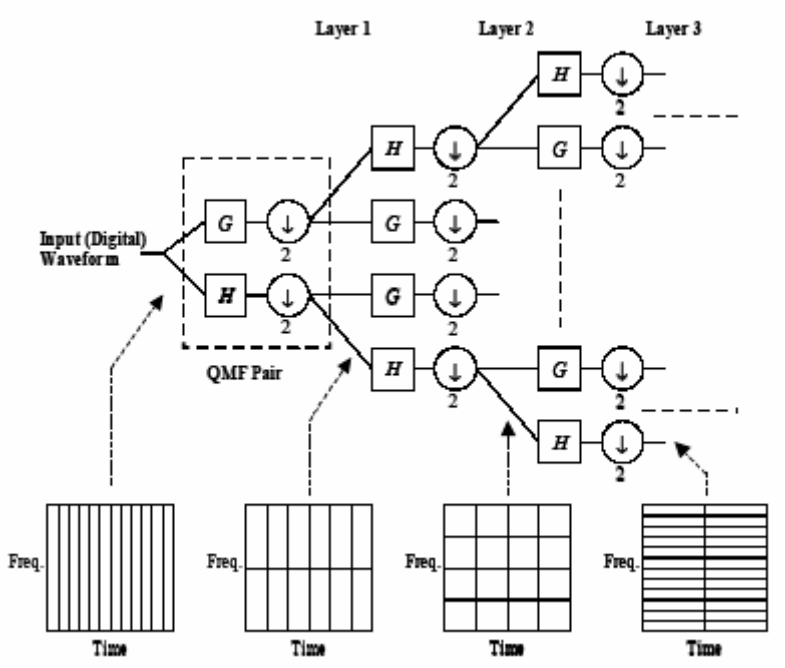

Figure 7. The Quadrature Mirror Filter Bank Tree (From Ref. 6.)

The received signal is first padded with zeros to contain $N_p = 2^L$ samples where L is the number of layers within the tree. A normalized input of one sample per second is assumed, with a signal bandwidth of $[0, \pi]$. Since each filter's output signal has half the bandwidth, only half the samples are required to meet the Nyquist criteria; therefore, these sequences are down sampled by two and the same number of output samples is returned. Each of the two resulting sequences is then fed into QMFB pairs, forming the next layer, where the process is repeated, and so on down the tree. The $l=L/2$ layer provides a good compromise in time and frequency resolution. The QMFB output strongly resembles the signal's periodic ambiguity function [6].

c. Cyclostationary Signal Processing

The detection technique which takes advantage of the bifrequency spectral analysis method is Cyclostationary processing. This model assumes that the input signal is periodically stationary and thus can measure the signal's periodic properties like modulation, sampling, and keying. These properties can be found through the results of the cyclic autocorrelation function and the spectral correlation density function. At certain frequency separations, known as the cycle frequency, a spectrally correlated signal is correlated with frequency-shifted versions of that signal. Finally, the spectral

correlation density (SCD) can be approximated by a method know as direct frequency smoothing whereby the spectral components of the signal are calculated and then the frequency components directly undergo a spectral correlation operation [6].

C. PREPROCESSING

The next stage is preprocessing the image passed in from the detection algorithm to create a feature vector to pass on to be classified by the neural network. The main goals of preprocessing are to extract the features of modulation within the image, to reduce the dimensionality, and to keep the feature vector output as unique and small as possible [6]. Generally, the preprocessing stage begins with cropping, but this is not necessary when the Quadrature Mirror Filter Bank is used. With QMFB, preprocessing is accomplished by thresholding, binarization, and finally reshaping the input matrix into a unique feature vector.

The preprocessing stage is discussed in depth in the following chapter.

D. NEURAL NETWORK CLASSIFICATION

The final stage is classification. Simply put, classification involves non-linear processing of the feature vector to put the data into a form in which the best decision can be made about what type of modulation was used to create the signal. This is accomplished by processing the feature vector received from the preprocessing stage through a three-layer perceptron neural network. These neural networks are well-suited to the classification stage because of their non-linear aptitude for learning and recognizing complex patterns.

1. Autonomous Classification Methods

Currently, expertly trained human operators are able to identify the signal parameters and determine the modulations received from the outputs of the time-frequency and bifrequency algorithms. In general, relying on a human to perform a task such as this adds considerable delay to the process. Classification would be significantly sped up if the classification stage were automated and performed by a fast computer. This way, signals could be classified in real time and the results would be useful in time-sensitive applications.

Currently, there are three major techniques used in the autonomous classification of signals. These techniques are energy detection, decision theory, and pattern

recognition. This study seeks out a new classification method based on the use of prior knowledge to continually train a hierarchical neural network to group similar signals into classes. A human programs a set of rules into the neural network that allow it to quickly group signals into classes. This greatly expedites the learning process and the accuracy of the classification results [6].

2. Recognized Modulations

a. FMCW Modulations

This neural network is trained to recognize several different modulations. One of the important modulations that can be recognized is Frequency Modulation Continuous Wave (FMCW), which is often used in the measuring of the range and range rate of a target. There are two linear frequency modulation parts of the waveform which have positive and negative slopes. FMCW is effective as a modulation for Low Probability of intercept signals because it spreads the transmitted energy over a large bandwidth which allows good range resolution. Also, its rectangular power spectrum makes non-cooperative signal interception complicated. FMCW is additionally a deterministic waveform which makes the return signal form predictable and resistant to undesired interference [6].

b. BPSK Modulations

Another type of modulation that is commonly used is the Barker-code Binary Phase Shift Keying (BPSK). This code consists of discrete time complex sequences of finite length with a constant magnitude. It results in a periodic ambiguity function with low side lobes relative to the main lobe. The code itself is not Low Probability of Intercept; rather it is simply a sequence used to perform the phase modulation. However, it is a solid benchmark to compare the classification techniques against [6].

c. Polyphase Modulations

Polyphase is a type of modulation with more than two phases that includes the Frank code and the P1 through P4 phase codes. The Frank code is similar to linear frequency modulation and Barker codes, but it has been successfully integrated into Low Probability of Intercept radars. It is derived from a step approximation to a linear frequency modulation waveform with M frequency steps and M samples per frequency. Much like Frank codes, the P1, P2, P3, and P4 codes are also derived from linear

frequency modulated waveforms. They are also made up of discrete phases of the linear chirp waveform, but they achieve lower side lobe levels. For all Polyphase codes, the time spent at any phase state is constant within the code period [6].

d. Polytime Modulations

The final modulation group is Polytime. This consists of the T1, T2, T3, and T4 codes which also aid in the estimation of stepped frequency modulations. Unlike Polyphase codes, these Polytime codes vary the time spent at each phase state throughout the code period. Polytime approximations improve in quality when the total number of phase states is higher. Unfortunately, this tends to complicate waveform generation by reducing the time spent at any particular phase state. The phase state durations change with time and the shortest duration sets the bandwidth [6].

3. Modulation Results

Dr. Phillip Pace of the Naval Postgraduate School tested these modulations against each other using set inputs. Table 2 shows an (non-optimized) example of using Quadrature Mirror Filter Bank detection-classification signal processing. The maximum neural network output is used for each classification vector, presented as a Confusion Matrix (CM). The first and second blocks show the results of testing the network with the training signals. The third block shows the results of testing the network with the training signals but with various signal-to-noise ratios. The fourth block shows the results when testing the network with the training signals with variations in the modulation parameters. The columns of this Confusion Matrix indicate the input modulation type, the rows show what was assigned by the neural network, and the diagonals give the percent chance that the proper modulation was chosen.

Table 2. QMFB Classification Results.

Training Signal CM	BPSK	FMCW	FRANK	P4	PT1
BPSK	**100%**	0%	0%	0%	0%
FMCW	0%	**100%**	0%	0%	0%
FRANK	0%	0%	**100%**	0%	0%
P4	0%	0%	0%	**100%**	0%
PT1	0%	0%	0%	0%	**100%**
Training Noise CM	BPSK	FMCW	FRANK	P4	PT1
BPSK	**100%**	0%	0%	0%	0%
FMCW	0%	**100%**	0%	0%	0%
FRANK	0%	0%	**100%**	0%	0%
P4	0%	0%	0%	**100%**	0%
PT1	0%	0%	0%	0%	**100%**
Test Modulation CM	BPSK	FMCW	FRANK	P4	PT1
BPSK	**80%**	95%	45%	43%	0%
FMCW	20%	**5%**	23%	18%	0%
FRANK	0%	0%	**10%**	23%	30%
P4	0%	0%	20%	**18%**	5%
PT1	0%	0%	3%	0%	**65%**
Test Noise CM	BPSK	FMCW	FRANK	P4	PT1
BPSK	**92%**	16%	1%	2%	0%
FMCW	8%	**72%**	3%	4%	0%
FRANK	0%	6%	**87%**	10%	6%
P4	0%	4%	7%	**83%**	1%
PT1	0%	1%	3%	1%	**93%**

E. SUMMARY

Chapter III described the general flow of the project stages. The reader should now be familiar with the stages of Low Probability of Intercept signal processing which include detection, preprocessing, and neural network classification. This chapter delved into detail about the detection and classification stages. The detection section discussed the general techniques used in LPI signal processing as well as the ELINT algorithms which can be used to detect the required modulations. The neural network classification section discussed the methods that can be used for classification, the particular modulation types supported by this process, and gave some experimental modulation results to offer a feel for how well each modulation type works.

The detection and classification stages were not covered by this research, but it is necessary to know how they function because they are directly linked to the preprocessing stage which is the main focus of this thesis. The detection stage provides the input to the preprocessor while the neural network classifier utilizes the output of the preprocessor to discern what modulations have been used.

Chapter IV will now fill in the details of the preprocessing stage which were omitted from this chapter.

IV. PREPROCESSING

A. INTRODUCTION

The main purpose of the preprocessing stage is to make a composite feature vector, from the input time-frequency image, to pass on to the neural network for classification. This process maintains the important properties of the modulation while reducing the dimensionality. The stages of preprocessing are explained below, they are: cropping, thresholding, binarization, and resizing the image. The preprocessing stage of the detection and classification of LPI signals through a Quadrature Mirror Filter Bank is implemented by the code developed for this thesis. This flow chart, Figure 8, provides an accurate visual aid for the general stages of preprocessing.

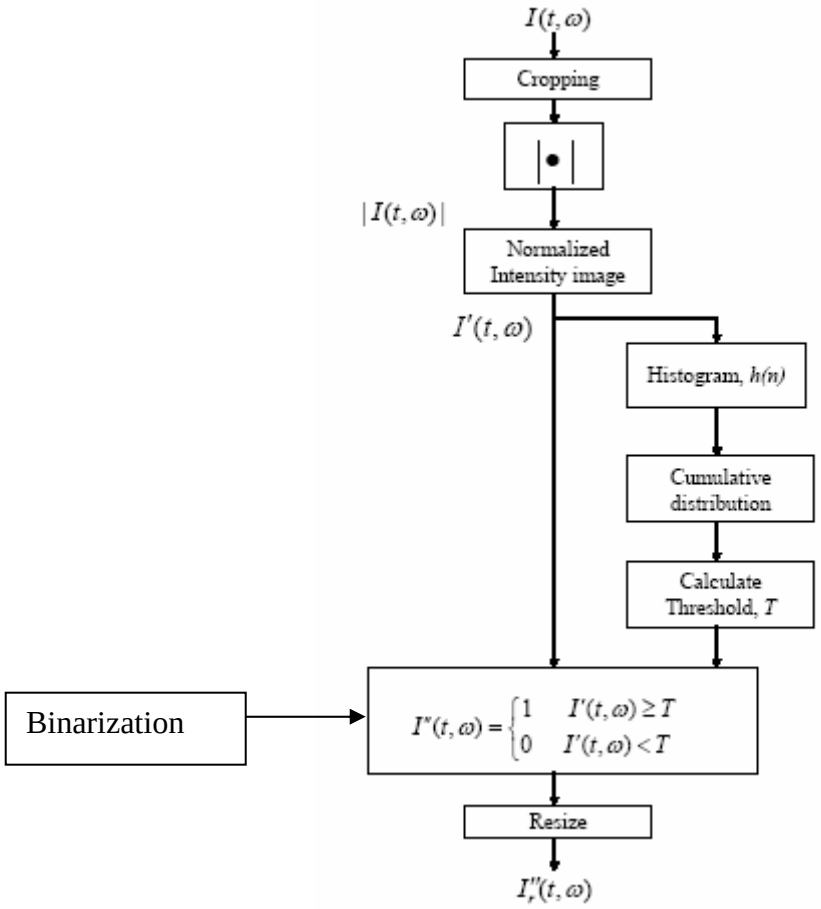

Figure 8. Preprocessing Flow Chart (After Ref. 8.)

Preprocessing of the Quadrature Mirror Filter Bank is very similar to that of the Wigner-Ville Distribution, except that image cropping is not necessary. The QMFB produces several time-frequency layers each with a different time and frequency resolution. For each progressing layer, the number of frequency bins is halved while the number of time bins is doubled. The number of pixels in the image, however, remains constant. The method used here differs even from conventional preprocessing techniques in that it uses an adaptive threshold computed by a cumulative distribution function on the normalized matrix.

B. CROPPING

1. Standard Cropping

Cropping of the input matrix, $I(t,w)$, is accomplished by extraction of a subset of the original matrix that contains the signal energy. The region that will be cropped is determined by features such as maximum signal duration, center frequency, and bandwidth. These parameters are all determined ahead of time. Though cropping is an essential process in the Wigner-Ville Distribution, the QMFB tree method differs in that cropping is not necessary [8].

2. Black-Cropping

After the time-frequency image has been input and cropped, there is frequently a section to the right of the image that contains “no signal”. This section is completely black and provides no additional useful information to the neural network. Black-cropping is the process of removing this part of the signal from the image in order to simplify classification.

C. THRESHOLDING

In order to prepare the image to be formatted with all ones and zeros, an appropriate threshold needs to be calculated to determine which bits will take on which values. To facilitate this, the cropped image, which is the original image in the QMFB case, needs to be normalized to the largest value in the matrix. The intensity levels, h(n), of the normalized image must then be put into a histogram using a number of bins that will provide reasonable resolution. In this case, 32 bins were used. The histogram may then be used in the computation of the cumulative distribution function (CDF) as follows [8]:

$$cdf\,(n) = \frac{\sum_{i=1}^{n} h(i)}{\sum_{n=1}^{N} h(n)} \tag{4.1}$$

These data are then compared to the CDF threshold to determine the system threshold. The CDF threshold is predetermined for the specific application; for the Quadrature Mirror Filter Bank tree, the CDF threshold C=0.9. This means that only the brightest 10% of pixels will be retained after thresholding; it differs from the typical CDF threshold of the Wigner-Ville Distribution process, which has C=0.8. The system threshold, T, is finally calculated by determining the bin value, n from the above equation, where $cdf\,(n) \geq C$ [8].

D. BINARIZATION

After a proper threshold has been computed, it is then used to convert the intensity image to all black or white pixels. These pixels are denoted by zeros (white) and ones (black) so the process is known as binarization. This process is effective at removing much of the noise and weak interference initially present in the intensity image [6].

For this project, the input values will range from 0 to 255. The threshold will be somewhere between these two boundaries. Each value in the matrix below the threshold will be changed to a zero, while the values above the threshold will be set to ones. This equation describes the computation made on each element of the normalized intensity matrix, $I'(t,w)$, to transform it into a binary image, $I''(t,w)$ [8].

$$I''(t,\omega) = \begin{cases} 1 & I'(t,\omega) \geq T \\ 0 & I'(t,\omega) < T \end{cases} \tag{4.2}$$

E. RESIZING

The final stage in preprocessing is the resizing of the matrix to form a feature vector. This is the vector which will be passed on to the neural classification network for a determination of the signal modulation. Generally, the image is reshaped using low pass filtering. It is then resampled and bilinearly interpolated to mitigate the undesired

effects of aliasing. The matrix is finally transformed into a feature vector of a length which is equivalent to the product of the numbers of rows and columns of the normalized intensity matrix [8].

F. SUMMARY

This chapter should provide the reader with all of the necessary details to comprehend how Low Probability of Intercept signals are preprocessed. It gives a description of all the steps involved in preprocessing which are: cropping, thresholding, binarization, and resizing. The figure in the introduction to this chapter shows the complete flow of this stage, which is a great supplement to the explanation of each step.

The chapters to follow will show and explain the different codes used which perform the preprocessing function of autonomous LPI detection and classification.

V. ALGORITHM PORTING

A. INTRODUCTION

1. Purpose

The main purpose of this research was to find out if Low Probability of Intercept signals could be autonomously detected and classified through the use of Electronic Intelligence algorithms on a reconfigurable computer. Some research had previously been done by Professor Phillip Pace of the Naval Postgraduate School on using these algorithms on a general purpose computer. His work successfully simulated the detection to classification sequence using the software simulation package MATLAB.

The work in this thesis concentrated on transferring work similar to that done by Dr. Pace in MATLAB to a reconfigurable platform. This involved modifying the MATLAB code he used to the specifications of this project. Then the code needed to be ported over to the SRC-6 C language so that it could be tested on the reconfigurable computer. In order to smooth this transition and to provide an intermediate benchmark, this code was also produced in the regular C language.

2. Procedure

For this implementation of the preprocessing stage, it was necessary to simulate data that might have been input from the Quadrature Mirror Filter Bank. This was done in all three versions of the code by generating a matrix filled with random numbers. For the scope of this project, it was assumed that layer five of the QMFB tree was used, thus the input image matrix always had the dimensions of 2^5 by 2^5, or 32 by 32. The expected input into the matrix was eight-bit integer values, which would range from 0 to 255. Each case used 32 bins to make a histogram of this input data. It was also assumed that the Cumulative Distribution Function (CDF) threshold was 0.9, which is standard for the Quadrature Mirror Filter Bank and means that only 10% of the brightest pixels with values above the CDF threshold are retained. After all of the necessary preprocessing was done in the code, an output feature vector containing the properly thresholded values was formed and ready to be sent to the Neural Classification Network.

B. MATLAB CODE

A complete listing of the MATLAB function used is located in Appendix A, along with the output generated by running it.

The code begins with a standard MATLAB function definition.

```
function preproc(layer, cthresh)
```

The name of this function file is preproc.m and the name of the function itself is also preproc, which is meant to affirm that this code deals with the preprocessing stage of the detection to classification sequence. There are two input parameters to this function, they are layer and cthresh. These allow the user to specify which QFMB layer and what Cumulative Distribution Function threshold to use, respectively. Since it is known that layer five is being used and that the CDF threshold is 0.9, a user can run this program by typing the following at the MATLAB prompt:

```
preproc (5,0.9)
```

This will produce the output shown in section 2 of Appendix A.

Within the body of the code, the first task was to define variables that would be used several times throughout the code in order to make the program more readable.

```
max_in=256;
num_bins=32;
len=2^layer;
```

The *max_in* variable represents one more than the maximum value that could be input into the matrix. It needed to be defined in this manner because the random function, shown below, generates a number which is at most one less than the variable by which it is multiplied. The number of bins to be used in the histogram is stored in *num_bins* and the dimensions of the input matrix are stored in *len* by raising two to the power stored in the user defined variable *layer*. Since layer five is being used, it is already known that the row and column dimensions defined by *len* are 32.

Using these variables, MATLAB can create a random input image matrix as follows:

```
img = floor(max_in*rand(len, len));
```

The random function, *rand()* in MATLAB, creates a matrix of size `len` by `len`, filled with random numbers between 0 and 1. In order to get these numbers into the desired range, they must each be multiplied by 256 which is `max_in`. The numbers produced here still have decimal precision which must be either truncated or rounded to leave just integer input values. This is done by utilizing the MATLAB function *floor()* which leaves a matrix of integers between 0 and 255 to be stored into the image variable `img`.

Once the input image matrix is formed, preprocessing of the data can begin. In order to find the threshold, the values in the image matrix must be stored into a histogram which then has its elements processed by a Cumulative Distribution Function. This is done easily in MATLAB as it provides high level built in functions which accomplish these tasks, a luxury not found in the C language.

The MATLAB function *hist()* makes a histogram of a vector when given a specified number of bins in which to store the data. The image matrix must be flattened into a vector using the MATLAB function *(:)* as shown below.

```
[bins, cent] = hist(img(:),num_bins);
```

Here, `bins` is a 32 element wide array representing the histogram bins. Each bin contains the number of elements from the input matrix whose value falls into the range of that bin. The vector `cent` is 32 bits wide as well and contains the value representing the center of each bin. This will be useful in finding the system threshold because once the bin is found that exceeds the CDF threshold, the system threshold is simply taken from the center value in `cent` corresponding to that bin in `bins`.

After determining the `bins` vector, the cumulative sums of its elements must be taken in order to compare them to the cumulative sum threshold, `cthresh`. The MATLAB function *cumsum()* creates a vector containing the cumulative sum of the

elements of the input array. Here, it will create an array of length 32 with each element containing the sum of array *bins* up to the index of the current component. For example, element number four of the cumulative sum vector will contain the sum of elements one through four of the *bins* vector. Since *ctresh* is a decimal value less than one, representing the percentage of the sum of elements the cumulative sum must reach, the elements of this new array must each be divided by the total number of elements in the input array with the result getting stored in a new vector named *cbins*. This sum can be taken through the MATLAB function *sum()* operating on the *bins* matrix flattened into a vector. To illustrate the power of programming in MATLAB, all of this is accomplished in the one line of code below.

```
cbins = cumsum(bins)/sum(bins(:));
```

Next, it is necessary to determine which decimal value in *cbins* exceeds the CDF threshold contained in *cthresh*. The MATLAB *find()* function, shown below, will compare each point of the *cbins* vector to the CDF threshold and report the indices of the vector where this threshold was reached. After this, the system threshold is computed by taking the minimum index where the CDF threshold was reached and taking the value of the bin center array, *cent*, which is found at this index. These two lines of code determine the threshold of the system.

```
ind = find(cbins >= cthresh);
thresh = cent(min(ind));
```

Once the system threshold has been found, the input image must be binarized. This is accomplished through the use of a MATLAB routine named im2bw(). This routine takes an input array and maps each component to a zero or a one based on whether or not it exceeds the threshold input, which must be between zero and one. To facilitate this, the threshold can be set to one and each component of the input image can be divided by the system threshold to normalize it about the binarization threshold of one.

```
feat = im2bw(img/thresh, 1);
```

The binarized image is then stored as the feature matrix, feat. This leaves only the final preprocessing step of matrix resizing to be accomplished. MATLAB simplifies the process of turning a matrix into a vector with its built-in function *(:)*. This vector can finally be transposed from a column vector to an easier to read row vector for output through utilization of the MATLAB operator ‘.

```
feat = feat(:)';
```

The final product is a binary feature vector ready to be sent to a Neural Network for classification.

C. STANDARD C CODE

With a working MATLAB implementation of the preprocessing algorithm, it becomes necessary to transfer that algorithm to the C language to determine if it is possible to program onto the SRC-6. Though C is still a high level language, it does not contain the myriad of engineering toolboxes that are included in MATLAB. Thus, many of the functions that were simple to implement in MATLAB by using these tools may present a greater challenge in C as these procedures need to be programmed using basic operations.

The C program, *preproc.c*, begins with the standard declarations of libraries and initial declarations of variables to be used throughout the program. A complete listing of the code, including these declarations, is found in Appendix B.

The variable layer is declared equal to 5 because this process uses the fifth layer of the QMFB tree. This allows use of the C function *pow()* to calculate the dimensions of the input matrix by raising two to the fifth power, yielding rows and columns of length 32.

```
len=pow(2,layer);
```

After the size of the input matrix has been determined, the programmer must assign random integers from 0 to 255 to the 32 by 32 input array. This will simulate potential input from the Quadrature Mirror Filter Bank Tree. Each element must be assigned its own random one at a time through the use of a nested for loop.

```
for(j=0; j < len; j++)
{
  for(k=0; k < len; k++)
    {
    image[j][k] = (int) (256.0*rand()/(RAND_MAX+1.0));
    i = image[j][k] >> 3;
    bins[i] += 1;
    }
}
```

While each element is being stored, it can immediately be placed in the proper bin during the same iteration of the nested *for-loop*. Since 32 bins has been chosen as the appropriate amount to store the data for the histogram, each bin has a range of $\frac{256}{32}$ or 8 numbers. This means that numbers 0 through 7 will go into bin 0, numbers 8 through 15 into bin 1, and so forth ending with numbers 248 through 255 going into bin 31. The simplest way to determine in which bin a specific element will belong is to divide the value of that element by eight, the resulting integer value will also be the appropriate bin number. It is important to minimize execution time wherever possible in this code and division by eight will take many clock cycles to execute. Fortunately, eight is a power of two, thus it is possible to right shift the value by three places which will execute much faster than a division by eight. Once the proper bin number is calculated, the program simply needs to increment the count of elements in that bin.

For visual reference, a histogram display has been added to the code to compare with the one generated by MATLAB. This is not, however, a critical part of the preprocessing procedure so it is not included for the timing analysis.

In order to binarize the input image, the system threshold must first be found. This must be accomplished, though, without use of the convenient engineering functions found in MATLAB. There is no *sum()* function in the C language, but the `sum` of elements can be found by taking the product of the input array dimensions. Similarly, no *cumsum()* function exists, but each element in the `bins` array can be progressively summed in a while loop and then compared to a CDF `cutoff` variable equal to the `sum` of input elements multiplied by the CDF threshold percentage. A while loop is better than a

for-loop in this case because it allows an early break of execution when the threshold *thresh*, which is initially zero, is found and becomes non-zero. The code for this *while-loop* is provided here.

```
i=0;
sum = len*len;
cutoff = sum*Cthresh;
while(!(thresh))
  {
    csum += bins[i];
    if(csum >= cutoff)
    thresh = 8*i + 4;
    i++;
  }
```

When the Cumulative Distribution Function threshold is reached, the system threshold becomes the center of the current bin, which is stored in the temporary variable *i*. It is calculated by multiplying the bin number by eight and then adding four. Once the system threshold is determined, the input image may be binarized by comparing each element within to the computed threshold value. This may be done through the use of a nested *for-loop* in which each element will be compared to the threshold yielding a binary result. If that element is less than the threshold, the result will be zero, otherwise it will be one. To help compact the output data, each set of 32 binary results is placed in one 32-bit wide integer value. Each bit is stored into this integer by shifting the binary comparison result an appropriate number of bits to the left and adding it to the integer. In total, 32 of these 32-bit integers are created which represent the binarization of all 1024 initial image values. These 32 integers are then placed into one 32 element array named *feature* in order to represent the feature vector which will be passed on to the Neural Classification Network for determination of the signal modulation.

```
for(j=0; j < len; j++)
  {
    for(k=0; k < len; k++)
    {
      feature[j] += (image[j][k]>=thresh) << 31-k;
    }
  }
```

D. SRC C CODE

1. Main Program

The main program named *main.c*, which facilitates execution of code on the SRC-6, is an ordinary C program which calls a subroutine that runs on the reconfigurable computer. Prior to the body of the C code, a function prototype for this subroutine must be defined. This subroutine will take two arrays of 64-bit integers, a pointer to a regular integer (32 bits), a pointer to a 64-bit integer, and one regular integer. It is a void function and will not return any values, except those passed by reference, to the main program.

```
void subr (uint64_t* , uint64_t*, int*, uint64_t*, int);
```

Following this prototype definition, the main body function is defined along with all of the variables required for execution of the program. The program *main.c* is listed in its entirety along with these definitions in Appendix C. One variable, however, needs to be declared in a special way. In order to be passed properly to the subroutine, the input image array needs to be declared in a format that is compatible with two-dimensional arrays on the SRC-6. This is accomplished by declaring a new variable type, called arr32, which is a 32 element wide array containing 64-bit integers. Creating a pointer to a variable of this type allows an array of these vectors to be declared in a way that the SRC-6 accepts.

```
typedef uint64_t arr32 [32];
arr32 *image;
```

In order to pass this image array and the feature vector to and from the subroutine, space must be allocated for them on the MAP. This is accomplished through the use of an SRC-6 allocation function name Cache_Aligned_Allocate as can be seen below. The Cache_Aligned_Allocate function must be passed the total size in bytes that must be allocated. For a 32 by 32 matrix of 64-bit integers, this is 32*32*8 = 8 kilobytes.

```
image = (arr32*) Cache_Aligned_Allocate(32*32*8);
feature = (uint64_t*) Cache_Aligned_Allocate(32*8);
```

Random numbers are again assigned to the input array to simulate an image being passed in from the Quadrature Mirror Filter Bank tree. This is done the same way as in the standard C program above. After defining the matrix values, it must be prepared to be sent across to the MAP for preprocessing. This command assigns a MAP processor and binds it to the subroutine that is about to be executed.

```
map_allocate (1);
```

Now the input *image* can be passed to the subroutine. The pointer to the *feature* vector must also be passed so that it can be retrieved when the function ends. The threshold variable *thold* is passed as well so that the user can verify that the correct threshold was computed when it is displayed in the program output, which is also found in Appendix C. The variable *tm* is passed to the subroutine so that the execution time of the program may be tracked and compared to other platforms. Finally, the *mapnum* variable must specify which MAP is being used for this subroutine.

```
subr(image, feature, &thold, &tm, mapnum);
```

Once the program returns from the subroutine, it prints the threshold value, the values of the feature vector, and the number of clock cycles used for execution of the subroutine to the user. Lastly, it frees the MAP processor with the following command and exits.

```
map_free (1);
```

2. Subroutine

The subroutine body contains local variable definitions just as the main program does. It also begins with a function header that must match the function prototype which preceded the main program body. One important constant to define for this routine is len with a value of 32. This is the size of the matrix dimensions and it is used often enough in loops and expressions that defining it as a constant makes the code much easier to read.

```
#define len 32

void subr (uint64_t Ain[], uint64_t Din[], int *threshold,
uint64_t *time, int mapnum)
```

This subroutine uses On Board Memory banks to store the values of the input image and also the output vector. OBM bank A must be declared as 2D in order to receive the two-dimensional array of the input image. OBM bank D is a one-dimensional array that will be used to store the output feature vector.

```
OBM_BANK_A_2D (A, uint64_t, len, len)
OBM_BANK_D (D, uint64_t, len)
```

After the OBM banks are declared, the data to be manipulated can be streamed into and out of them. The input image matrix is streamed into OBM bank A as follows.

```
DMA_CPU (CM2OBM, A, MAP_OBM_stripe(1,"A"), Ain, 1,
len*len*sizeof(uint64_t), 0);
wait_DMA (0);
```

The SRC-6 includes a loop-flattening technique shown below which reduces the dependencies on the nested loop variables *j* and *k*. This allows the MAP processor to execute the loop in less clock cycles, which will likely result in significant timing gain over other platforms running the same algorithm. Inside the loop, elements are separated into their respective bins in the same manner as in the standard C code.

```
for(xs=0; xs < len*len; xs++)
{
      cg_count_ceil_32(1, 0, xs==0, len-1, &k);
      cg_count_ceil_32(k==0, 0, xs==0, SZ, &j);
      i = A[j][k] >> 3;
      bins[i] += 1;
}
```

The next step involves comparing the CDF threshold *cutoff* to the ongoing cumulative sum until the system threshold is found. This is accomplished no differently than in the standard C code above. Once this threshold is found, the loop-flattening process can again be used to speed up the process of storing each binarized bit into the proper place in an integer element of the output feature vector.

```
for(xs=0; xs < len*len; xs++)
{
      cg_count_ceil_32(1, 0, xs==0, len-1, &k);
      cg_count_ceil_32(k==0, 0, xs==0, SZ, &j);
D[j] += (A[j][k]>=thresh) << 31-k;
}
```

Finally, the output feature vector, which is contained in OBM bank D, can be streamed out into variable Din to be passed on to the Neural Classification Network, or as in this case, back to the main program. The subroutine now passes all of the necessary parameters back to the main program and relinquishes control.

```
      DMA_CPU (OBM2CM, D, MAP_OBM_stripe(1,"D"), Din, 1,
len*sizeof(uint64_t), 0);
      wait_DMA (0);
```

3. Make File

The files described above are not compiled with the general C compiler; rather they are compiled by using a Make File as described in Chapter II. The entire Make File is listed after the other SRC-6 files in Appendix C. However, there is very little of it that pertains specifically to these programs. In fact, the only changes that must be made to the generic Make File template are specifications of the input file names and the desired name of the output executable program. Those changes which were made to the file are shown below.

```
# ----------------------------------------
# User defines FILES, MAPFILES, and BIN here
# ----------------------------------------
FILES             = main.c

MAPFILES    = preproc.mc

BIN         = preproc
```

E. SUMMARY

In this chapter, the three different variations of code found in the appendices were completely explained. These programs all perform the preprocessing stage of the detection to classification sequence and they are useful tools to assist in the autonomous detection and classification of Low Probability of Intercept signals. The first section covered a MATLAB program adapted from the code of Professor Phillip Pace which

successfully performs this function. The next section explained the translation of this code into the standard C language where it can be compiled and executed on any personal computer or Unix-based system with the appropriate compiler. The last section described the C code which is specific to MAP execution on an SRC-6 machine; it detailed the two programs needed to run code on the general purpose microprocessor as well as the MAP processor and it also briefly described the Make File which allows compilation of this code.

The reader should now have a firm understanding of how the preprocessing stage works as well as how it can be programmed for execution on a variety of computing platforms. Specifically, one should comprehend how to create a preprocessing program which will run quickly and efficiently on the SRC-6. The following chapter will analyze the actual timing of the three programs which were described above. It will seek to determine which platform executes this code in the fastest and most efficient manner.

VI. PERFORMANCE ANALYSIS

A. INTRODUCTION

It has now been illustrated that the presented ELINT algorithms can be successfully programmed onto standard computer systems, as well as reconfigurable computer systems, for the purpose of detecting and classifying Low Probability of Intercept signals. This chapter compares the three solutions described above to determine which computing platform has the best performance. Timing data has been taken of the run times for the MATLAB program on a Windows PC, the standard C program in a Linux environment, and the SRC C programs on the MAP.

Only the parts of these programs dealing specifically with the preprocessing of data were timed. The changes made to the code to acquire these data are also briefly detailed in this chapter.

B. EXECUTION TIMES

1. MATLAB

The first set of timing data was extracted from several executions of the MATLAB preprocessing program, *preproc.m*. These executions were run on a Microsoft Windows XP machine with an Intel Pentium 4 processor running at 3 GHz with 512 MB of RAM. In order to obtain the timing data, it was necessary to make several minor modifications to the code.

First of all, MATLAB timing commands were used at the beginning and end of the preprocessing section and the difference was taken to determine execution time. The MATLAB function *cputime* returns the current system clock time in seconds, thus the time difference was taken by subtracting a time already recorded from the current *cputime*.

```
y=cputime;
--Preprocessing Loop omitted--
time=cputime-y;
```

The problem with the timing data given by MATLAB is that the resolution is not small enough. By experimentation, it was determined that the shortest period of time

which could be detected by the MATLAB timing function was approximately fifteen milliseconds. The next lower and higher results that would be output by MATLAB were zero seconds and approximately thirty milliseconds respectively. Also, MATLAB only shows these results to three decimal places, thus the user is given no granularity below the millisecond.

This presented a major problem because an efficient implementation of the preprocessing procedure takes relatively few clock cycles to execute. With a small number of layers, total execution time is often much less than one millisecond. This obstacle was circumvented by repeating execution of the preprocessing stage until enough time was taken up for MATLAB to produce timing data with reasonable resolution.

The next modifications to the code were the additions of a loop variable and a *for-loop*. The *for-loop* was placed around the section of code which handles the preprocessing of the input data. The loop variable was used to dictate how many times the *for-loop* would be run. The implementation of these changes is shown below.

```
for p=1: # ,
--Preprocessing Procedure omitted--
end
```

In order to more completely demonstrate the timing of the preprocessing procedure, the program was run on data representing output from several different layers of the Quadrature Mirror Filter Bank tree. Thus, the final change to the code was in the function call itself. The program was written as a MATLAB function which allows certain parameters to be declared by the user when calling it. The two parameters that *preproc.m* allows to be specified at runtime are the desired QMFB layer and the CDF threshold as can be seen below.

```
function preproc(layer, cthresh)
```

With this functionality in place, it was possible to run the code on every QFMB layer between layer three and layer ten. On each layer, a number of execution loops was experimentally determined which increased the total execution time to several tenths of a

second, thus quelling any concerns of a lack of resolution. Once the proper number of loops was found the program was executed ten times, and the timing data were tracked for every run. After the ten runs were complete, the average execution time of those runs was taken. Finally, the adjusted average was found which consisted of dividing the average by the number of loops that were executed. This gave the actual time, with adequate resolution, that it took for one execution of the preprocessing procedure to complete.

Table 3. MATLAB Execution Timing Data

	Run	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*Avg*	*Adj.*
Layer	**Loop**	Time (s)											*Avg (μs)*
10	*1*	0.3130	0.3280	0.3280	0.3130	0.3280	0.3120	0.3120	0.3120	0.3120	0.3120	0.3170	**317000**
9	*5*	0.4060	0.4060	0.3910	0.4060	0.4060	0.4060	0.4060	0.4060	0.3910	0.3900	0.4014	**80280**
8	*20*	0.4220	0.4220	0.4220	0.4220	0.4220	0.4220	0.4220	0.4060	0.4220	0.4060	0.4188	**20940**
7	*80*	0.4530	0.4540	0.4370	0.4370	0.4380	0.4370	0.4220	0.4220	0.4370	0.4380	0.4375	**5468.75**
6	*320*	0.5780	0.5940	0.5620	0.5620	0.5630	0.5620	0.5630	0.5620	0.5630	0.5470	0.5656	**1767.5**
5	***500***	**0.4210**	**0.4540**	**0.4370**	**0.4380**	**0.4370**	**0.4380**	**0.4380**	**0.4370**	**0.4220**	**0.4370**	**0.4359**	**871.8**
4	*1000*	0.6880	0.6560	0.6250	0.6410	0.6090	0.6250	0.6100	0.6100	0.6250	0.6250	0.6314	**631.4**
3	*1000*	0.5940	0.5780	0.5630	0.5930	0.5780	0.5790	0.5780	0.5620	0.5630	0.5620	0.5750	**575**

Table 3 above shows these timing data for every layer. The two leftmost columns tell what QMFB layer is being operated on and how many loops are being executed, respectively. The top row tells to which of the ten runs of the program the timing data in its column corresponds. The final two columns give the average and the adjusted average run time for each layer.

As can be seen in the table, layer ten has so many elements ($2^{10} * 2^{10} = 2^{20}$) that it already takes several hundred milliseconds to execute only once, thus it need not be looped more than one time. However, each layer below ten executes progressively faster and loops become increasingly necessary. The general pattern shown by the chart is that each successively lower layer takes approximately four times as many loops to obtain a reasonable total execution time on the order of tenths of a second. This is intuitive as each lower layer has one fourth the number of elements as the preceding layer to operate on. This same effect trickles down to the adjusted average times as each lower layer realizes a speedup of approximately four times when compared to the layer before it. This effect can be better understood visually from the following graph, Figure 9.

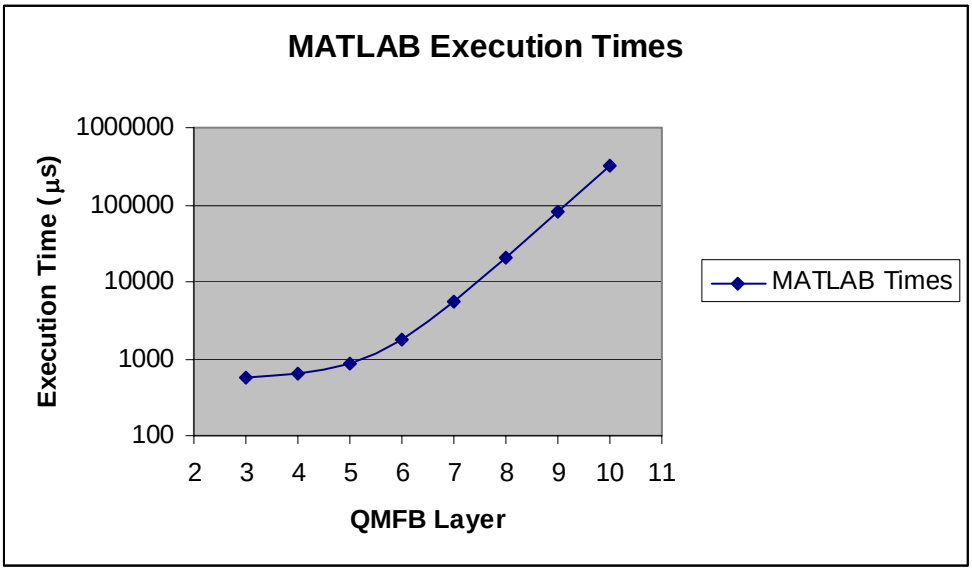

Figure 9. MATLAB Execution Times

Oddly, this speedup approaches somewhat of a plateau near layer five which, at 871.8 microseconds, is twice as fast as layer six, but neither layer four nor layer three realize much of an improvement over the layer five results. This could be caused by several possible factors. One possibility is that there is not enough data present at these low layers to take advantage of the cache, so the compulsory cache misses dominate the overall data processing. Another explanation may stem from the fact that MATLAB is an interpreted language rather than a compiled one. As such, the amount of overhead involved in interpreting each command at run-time could account for the majority of the execution time.

2. Standard C

Next, similar timing data was extracted from the program being run in the standard C language. This was run on a Linux machine with two Intel Xeon processors running at 2.8 GHz with 2 GB of RAM. Just as the MATLAB code needed to be modified, there were a number of changes that needed to be made to the C code in order for it to display accurate timing data. Within the code, new variables were declared to keep track of the time that had elapsed.

```
time_t begin, end;
double elapsed;
```

The *begin* and *end* variables are of type time_t which is a specific time data type defined in the time.h library. These were used to store the times at the beginning and end of the preprocessing procedure. The double precision floating point number *elapsed* was used to take the difference between these two times to alert the user of the total run time. These new variables were implemented as shown below.

```
begin = clock();
--Preprocessing Loop omitted--
end = clock();
elapsed = (double) (end-begin)/CLOCKS_PER_SEC;
printf("%.3f\t",elapsed);
```

The function *clock()* defined by time.h returns the number of clock cycles that have occurred since a predetermined moment. The last element used from the time.h library was the constant *CLOCKS_PER_SEC* which stores the number of processor clock cycles which occur in one second. When the difference between the beginning and end time is divided by this constant and the result is converted to the double precision data type, the total preprocessing time elapsed is given in seconds. This is how the variable *elapsed* was determined.

The final line details how the time information was displayed to the user. For two reasons, all other program outputs were eliminated. First, this single numerical output followed by a tab facilitated the use of a Linux shell script to run the program multiple times and send the timing data directly to a file which could then be fed into a spreadsheet. Additionally, sending data to standard output (the terminal screen) takes up extra time and the program runs faster without these extraneous procedures, which are now unnecessary since that data has already been collected and displayed in Appendix B. The elimination of superfluous outputs was common across all three programs for the collection of timing data.

It was again determined by trial that the C program produced times of unacceptably large resolutions much like the MATLAB code. It was found that the shortest amount of time that could be measured by the C timing functions was ten milliseconds. Once again, it was necessary to loop the preprocessing section of the code

several times to find adjustable execution times of appropriate granularity. Thus, the next major change to the C code was the addition of the looping variables *loops* and *p*.

```
long int loops= # , p;
```

Specified prior to compilation by the programmer, *loops* contained the number of times to execute the preprocessing procedure in order to obtain a run time of hundreds of milliseconds. In the *for-loop* surrounding this procedure, *p* was used as the counter which was incremented until *loops* was reached.

```
for (p; p < loops; p++)
{
--Preprocessing Procedure omitted--
}
```

The final addition to the code was a constant *arlen* which was declared before the main function to define the dimensions of the input and output arrays. This was necessary for the programmer to change according to the current QMFB layer being operated on.

Once all of these changes were made, it was possible to extract the timing data from layer ten down through layer three. These are displayed in Table 4.

Table 4. Standard C Execution Timing Data

	Run	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*	*10*	*Avg*	*Adjusted*
Layer	**Loops**	Time (s)											*Avg (μs)*
10	*50*	0.55	0.56	0.56	0.56	0.56	0.55	0.57	0.57	0.57	0.57	0.562	**11240**
9	*200*	0.56	0.56	0.57	0.57	0.56	0.56	0.56	0.55	0.56	0.56	0.561	**2805**
8	*800*	0.56	0.56	0.57	0.56	0.56	0.59	0.56	0.56	0.57	0.57	0.566	**707.5**
7	*3200*	0.57	0.57	0.58	0.56	0.57	0.56	0.56	0.55	0.56	0.57	0.565	**176.5625**
6	*12800*	0.57	0.56	0.58	0.59	0.57	0.58	0.58	0.58	0.59	0.58	0.578	**45.15625**
5	***51200***	**0.58**	**0.57**	**0.55**	**0.55**	**0.54**	**0.57**	**0.55**	**0.56**	**0.54**	**0.55**	**0.556**	**10.85938**
4	*204800*	0.55	0.6	0.66	0.56	0.59	0.6	0.56	0.59	0.58	0.61	0.59	**2.880859**
3	*819200*	0.64	0.63	0.66	0.68	0.58	0.54	0.56	0.67	0.65	0.56	0.617	**0.753174**

This is laid out in a very similar manner as the MATLAB timing data. The top row notifies the viewer as to which of ten executions is being described. The left columns again denote which layer was being dealt with and how many loops were used

to achieve a run time in the desired range. It is clear that the adjusted average times, given here in microseconds, are consistently sped up by a factor of approximately four in each lower layer. As evidenced by the following chart, this pattern does not seem to break down at lower layers, suggesting that the C language handles this code in a more efficient manner. Additionally, the C code is compiled which means that it will not have to deal with issues such as interpretation overhead at low layers.

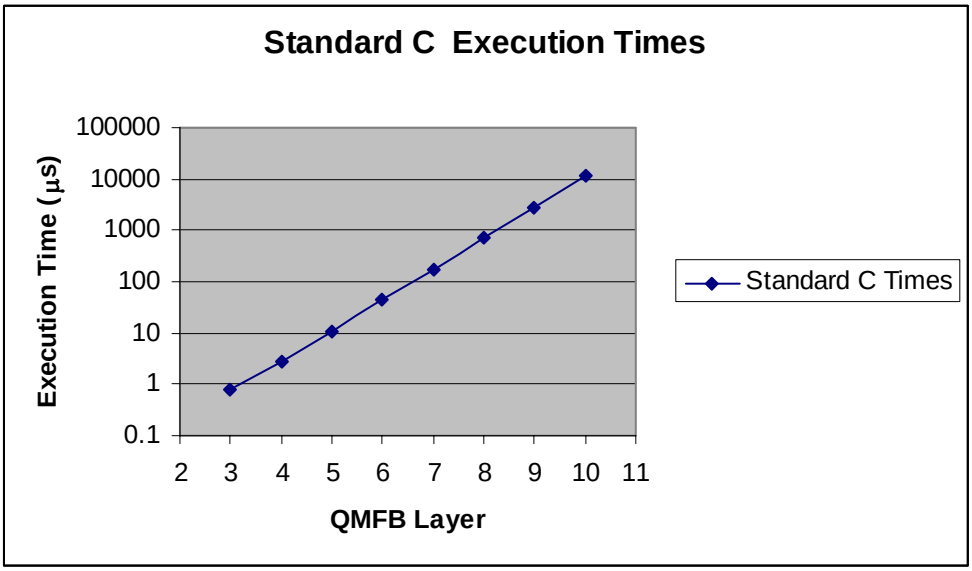

Figure 10. Standard C Execution Times

3. SRC-6

The final timing results came from execution of the preprocessing code on the SRC-6. It did not take as many changes to the original code to acquire these data as it did on the other platforms; there was already code included which determines the number of clock cycles taken to execute a section of the program and then displays that number to the user. In the original program, an integer variable `tm` was declared to store the number of clock cycles for display. This variable was passed to the subroutine where the *read_timer*() function was used before and after execution of the preprocessing procedure. The difference between these readings was stored into a `time` variable to be passed back into `tm` as follows.

```
read_timer (&t0);
      --Preprocessing Procedure omitted--
      read_timer (&t1);
      *time = t1 - t0;
```

Another convenient factor in extracting timing data from the SRC-6 was that resolution concerns were eliminated. Since the MAP returns timing data in the number of clock cycles executed, there was no need to manipulate the time collection algorithm to attain times with microsecond resolution. Thus, it was not necessary to iterate the preprocessing stage multiple times in a for-loop; each layer executed the procedure one time and output the number of clock cycles as shown in Table 5 below.

Table 5. SRC-6 Execution in Clock Cycles

Run	1	2	3	4	5	6	7	8	9	10	Average
Layer	Clock Cycles										
10	16778645	16778645	16778645	16778645	16778645	16778645	16778645	16778645	16778645	16778645	**16778645**
9	4195221	4195221	4195221	4195221	4195221	4195221	4195221	4195221	4195221	4195221	**4195221**
8	1049237	1049237	1049237	1049237	1049237	1049237	1049237	1049237	1049237	1049237	**1049237**
7	262677	262677	262677	262677	262677	262677	262677	262677	262677	262677	**262677**
6	66005	66005	66005	66005	66005	66005	66005	66005	66005	66005	**66005**
5	***16821***	***16821***	***16832***	***16832***	***16832***	***16821***	***16821***	***16821***	***16821***	***16832***	***16825.4***
4	4539	4539	4539	4539	4506	4506	4506	4517	4517	4517	**4522.5**
3	1448	1448	1448	1415	1415	1415	1437	1437	1437	1426	**1432.6**

It is interesting to note that the SRC-6 continues the trend of speeding up by about four times with each lower layer of the Quadrature Mirror Filter Bank. Also, the number of clock cycles appears to become much more stable at higher layers. This is curious and may be due to a phenomenon such as inaccuracies in time measurement. At lower layers, there is less data to be measured, so these inaccuracies are conspicuous. However, there is so much data to be processed at higher layers that the imprecision of each timing measurement becomes veiled. The number of clock cycles executed with respect to QMFB layer can be more clearly seen in Figure 11.

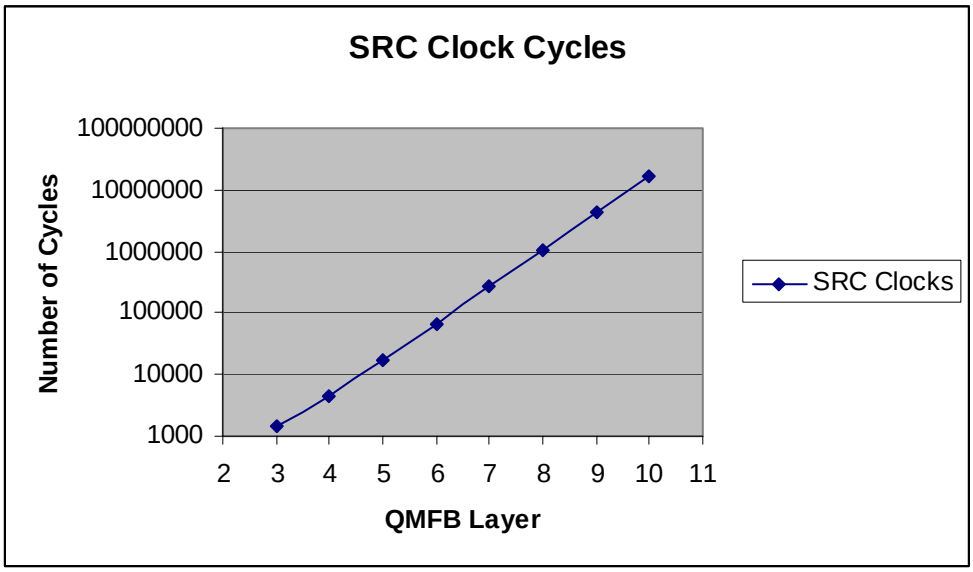

Figure 11. SRC-6 Clock Cycles Executed

To attain the actual timing data in standard time units, it was necessary to divide the number of clock cycles by the number of clock cycles that occur in one second, or the processor speed. It is known that the MAP executes at approximately 100 MHZ or 100 million cycles per second. Thus, the number of seconds taken for each execution was found by dividing the number of clock cycles by 100 million. This provided resolution down to ten nanoseconds, which was more than adequate for the program being executed. These times are also provided here in Table 6 and then shown in the graph, Figure 12, to follow.

Table 6. SRC-6 Execution Time Data

Run	1	2	3	4	5	6	7	8	9	10	Average (µs)
Layer	Time (s)										
10	1.68E-01	1.68E-01	1.68E-01	1.68E-01	1.68E-01	1.68E-01	1.68E-01	1.68E-01	1.68E-01	1.68E-01	**167786.5**
9	4.20E-02	4.20E-02	4.20E-02	4.20E-02	4.20E-02	4.20E-02	4.20E-02	4.20E-02	4.20E-02	4.20E-02	**41952.21**
8	1.05E-02	1.05E-02	1.05E-02	1.05E-02	1.05E-02	1.05E-02	1.05E-02	1.05E-02	1.05E-02	1.05E-02	**10492.37**
7	2.63E-03	2.63E-03	2.63E-03	2.63E-03	2.63E-03	2.63E-03	2.63E-03	2.63E-03	2.63E-03	2.63E-03	**2626.77**
6	6.60E-04	6.60E-04	6.60E-04	6.60E-04	6.60E-04	6.60E-04	6.60E-04	6.60E-04	6.60E-04	6.60E-04	**660.05**
5	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	**1.68E-04**	***168.254***
4	4.54E-05	4.54E-05	4.54E-05	4.54E-05	4.51E-05	4.51E-05	4.51E-05	4.52E-05	4.52E-05	4.52E-05	**45.225**
3	1.45E-05	1.45E-05	1.45E-05	1.42E-05	1.42E-05	1.42E-05	1.44E-05	1.44E-05	1.44E-05	1.43E-05	**14.326**

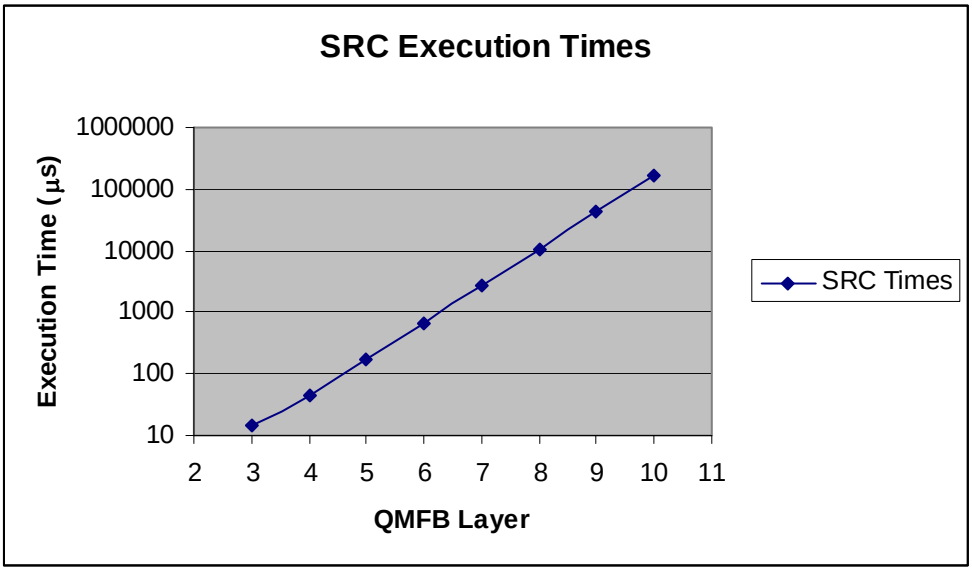

Figure 12. SRC-6 Execution Times

C. COMPARISON

Having accumulated timing data from multiple sources on the preprocessing stage of the detection to classification sequence, it was necessary to compare those data to determine which of these platforms offers the best environment for the autonomous classification of Low Probability of Intercept signals. The next chart summarizes the timing data obtained from all three methods.

Table 7. Comparison of Average Times

Layer	MATLAB Avg (μs)	C Avg (μs)	SRC Avg (μs)
10	317000	11240	167786.5
9	80280	2805	41952.21
8	20940	707.5	10492.37
7	5468.75	176.5625	2626.77
6	1767.5	45.15625	660.05
5	**871.8**	**10.85938**	**168.254**
4	631.4	2.880859	45.225
3	575	0.753174	14.326

Table 7 makes it very clear that the generic C language provided the fastest implementation by far. The SRC-6 times are average, while the MATLAB times are exceedingly slow, especially when approaching the lower layers. For Quadrature Mirror Filter Bank layer five, which was the major focus of this research, the C language implementation was able to execute the preprocessing procedure in only 10.9 microseconds. The SRC-6 took 168.2 microseconds, while MATLAB took an average of 871.8 microseconds to execute the code. This illustrates that at the most significant layer, the standard C code is 15.4 times faster than MAP execution, and 80 times faster than the MATLAB implementation. The following chart provides a good visual delineation of the relationship of the execution times across platforms.

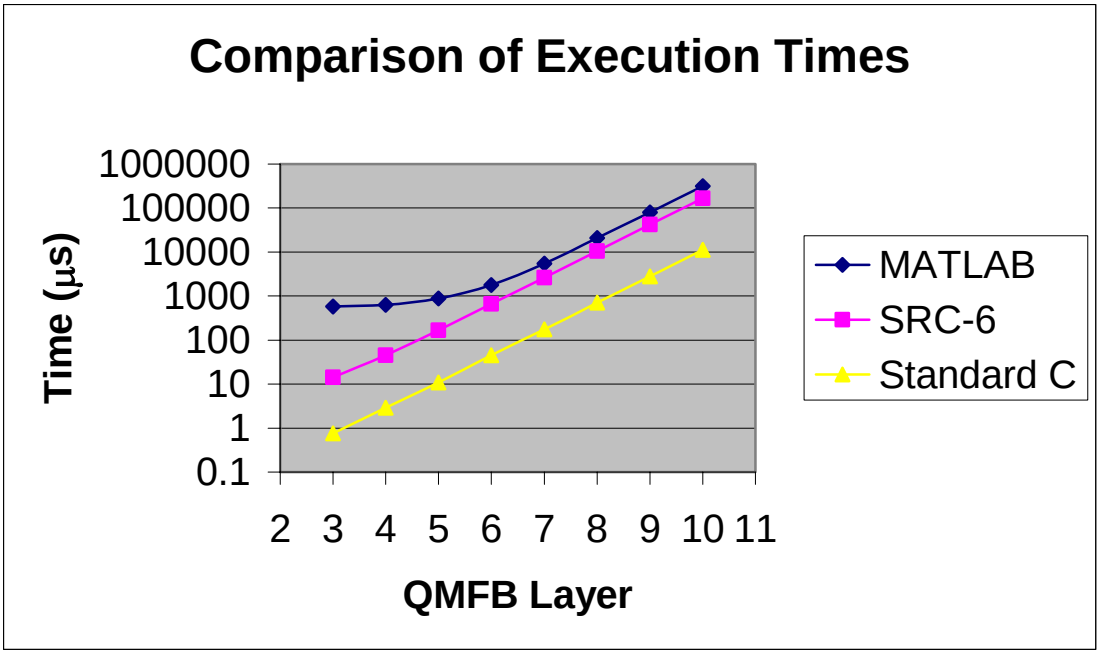

Figure 13. Preprocessing Procedure Execution Times

Figure 13 shows the execution times for all three solutions on a logarithmic scale. It can be seen here that at higher layers, all three implementations were achieving comparable significant speed ups. However, at lower layers the MATLAB code shows clearly that it reaches the peak of its performance potential and begins to lag even further behind the other two solutions. The graph also suggests that at higher layers, the SRC-6 provides performance relatively closer to that of MATLAB, and though it lags the C implementation at the same rate for all layers, the SRC-6 appears relatively closer to the performance of the standard C language as MATLAB approaches its maximum performance.

The relative lack in performance of the SRC-6 implementation when compared against a standard C program was a disappointment. The initial reason that running this algorithm on the SRC-6 was desired was because of the efficient use it makes of processing hardware relative to a personal computer. Though it has a slower clock cycle, it can usually make up the speed through efficiency when working with large data sets. One reason for the performance lag may have been that the fifth layer, with only 1024 elements was not a large enough set of data to realize the advantage of reconfigurable computing. However, the standard C implementation continued to outperform the SRC-6

at a consistent pace all the way up through layer ten, which has 2^{20} or 1048576 elements. Another sobering possibility is that the SRC-6 may have just been overwhelmed by the sheer processing power of the Linux machine on which the C program was run. The Intel Xeon processors in the Linux system operate at 2.8 GHz, which is 28 times faster than the 100 MHz clock of the SRC-6. Furthermore, if the two processors are working together constructively, they may be realizing an even more impressive speedup.

D. SUMMARY

This chapter analyzed the efficiency and performance in time of the three preprocessing solutions described in the previous chapter. These were: a MATLAB program, a standard C language program, and an SRC-6 based C program. First, this chapter described the changes that needed to be made to the code for each one of these implementations in order to obtain the timing information. Then, it showed what timing data was extracted from several iterations of each program on Quadrature Mirror Filter Bank layers three through ten. Finally, it compared all three timing sets to one another to determine which handles the preprocessing procedure the fastest. For the data sets generated in this work, it was undeniable that the standard C program provides the best solution if time is the most important factor.

THIS PAGE INTENTIONALLY LEFT BLANK

VII. CONCLUSIONS

A. SUMMARY

The goals of this thesis were twofold. It was necessary to determine whether ELINT algorithms for the detection and classification of Low Probability of Intercept signals could be coded for execution on a reconfigurable computer. If it was indeed possible, the second goal would become to ascertain what level of performance could be achieved when compared to commodity computing solutions.

This thesis began by introducing the major concepts pertaining to the research. It was necessary to provide an overview of the SRC-6. Understanding the hardware and software available on this system allowed the reader to follow the developed implementation much more easily. Next, the details of the specific project stages were explained. This detailed the overall detection to classification sequence for LPI emitters and included detection through established ELINT algorithms, preprocessing of the input Quadrature Mirror Filter Bank time-frequency image, and finally classification of the system by a neural network. After this, the preprocessing stage was narrowed in on because that stage of the sequence was the focal one for this research. All of the aspects of preprocessing were covered, which were: cropping, thresholding, binarization, and then resizing into a feature vector.

Once the reader had obtained a working knowledge of the system and the procedure to be performed, it would become possible to explain the implementations of the algorithms described above that were created for this thesis. This explanation began with a modification of a preprocessing procedure developed in MATLAB by Professor Phillip Pace of the Naval Postgraduate School. Then it illustrated how this code was translated for use in the conventional C language. Finally, the code was ported over to the SRC-6 using its C language interface. After all of the code had been written and debugged, the results from each implementation could be compared against one another to determine how well the SRC-6 could perform against the more common solutions.

It was found that, indeed, ELINT algorithms for the process of autonomous detection and classification of LPI signals can be coded onto a reconfigurable computer.

This is proven by the provided SRC-6 code which successfully performs the preprocessing stage of this procedure. As for the performance achieved, the SRC-6 clearly provides a significant advantage over using the MATLAB solution. However, for the data tested in this work, the SRC-6 still lags far behind the speed of the standard C language implementation.

B. PROBLEMS ENCOUNTERED

One major problem that had to be dealt with was the timing resolution on the MATLAB and standard C systems. There are apparently solutions available to both of these problems on the internet. There are said to be system files that can be compiled and placed in the MATLAB directory to improve timing granularity to the microsecond level. Also, there is speculation about a Linux kernel patch which provides microsecond timing resolution for C programs [9]. Neither of these solutions came to fruition as the kernel patch would require root access to the lab Linux system to install and the MATLAB program would not compile properly. Thus it was necessary to estimate high-resolution timing through the use of loops and averaging.

Another somewhat disappointing problem was the relative lack in performance of the SRC-6 implementation when compared against a standard C program. This may have been due to using a data set that was too small to notice a significant adavantage, but this is unlikely as the code was tested on a range of data set sizes. More than likely, the SRC-6 simply does not yet possess the resources to compete with a system that has the processing power of the Linux machine used in this research.

C. RECOMMENDATIONS FOR FUTURE WORK

Clearly, if these two problems could be solved, it would present an opportunity for more precise data and possibly even more desirable timing results for the SRC-6. For a more precise timing analysis, it may be helpful to find and install the aforementioned system patches which obtain microsecond timing.

It may also be possible to improve the relative performance of the preprocessing algorithm running on the SRC-6. The size of the data set used here was an expected drawback to using the reconfigurable computer, perhaps an optimal image size could be found at which point the SRC-6 implementation performs better than the generic C

language program. Furthermore, this solution did not seek to optimize the SRC-6 hardware. There are tools available such as programming macros in VHDL, this could potentially build a more efficient solution than the one presented here.

Finally, this solution by itself does not take into account the inherent parallelism of the reconfigurable computer. This thesis is part of a larger project that seeks to run all three stages of the detection to classification sequence in parallel. This means that while the preprocessing stage implemented here is working on one set of data, the detection stage will be processing a new input, and the Neural Network classifier will be processing the last output of the preprocessing stage, all simultaneously.

All programs and their outputs are given in the appendices to follow.

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX A. MATLAB CODE

A. MATLAB CODE

```
function preproc(layer, cthresh)

max_in=256;     %Define the maximum input value
num_bins=32;    %Define number of bins to use for the histogram
disp(strcat('QMFB layer: ',num2str(layer)));
disp(strcat('CDF Threshold: ',num2str(cthresh)));
len=2^layer;    %Define row and column sizes of input matrix

disp(' ');
disp('Random input image:');
%Create a random input image to above specifications
img = floor(max_in*rand(len,len));
disp(img);

%Perform preprocessing procedure

%Create a histogram of the values in the input matrix
[bins,cent] = hist(img(:),num_bins);
%Show the histogram bar plot

% figure(1)
% hist(img(:),num_bins)
% title('Data Distribution')
% xlabel('Bin Ranges'); ylabel('Number of Elements in Range');

%Take the cumulative sum of the elements throughout the bins
cbins = cumsum(bins)/sum(bins(:));
%Find the point where this sum exceeds the CDF threshold
ind = find(cbins >= cthresh);
%The first point where the CDF threshold is exceeded
%   becomes the system threshold
thresh = cent(min(ind));
disp(' ');
disp(strcat('Image Threshold: ',num2str(thresh)));
%Use the threshold to decide which elements of the input image
%   become ones and which elements become zeros.
feat = im2bw(img/thresh,1);
disp(' ');
disp('Binarized Feature Matrix:');
disp(feat);

%End of preprocessing procedure

%   Turn the result matrix into a feature vector by concatenating
%   all of the columns to make it one-dimensional.
feat = feat(:)';
disp(' ');
disp('Feature Vector:');
disp(feat);
```

B. OUTPUT

>>preproc(5,0.9)

QMFB layer:5
CDF Threshold:0.9

Random input image:
Columns 1 through 23

```
243  113  209   72  112  155  12   188  216  180  187    84  156  121
248  153  168  197  124  185  168  187  247

59   238  169  120  127    4   20  175   94  124  108   122   18  231
91   242   54   80  254  102  221  105  210

155  119   87   16   54    4  164   88  158   29  246   152   80  115
12    73  154  163   95   91  145  102   81

124  107   74  253  164   48   48   42  187  170   18    41  155   205
193  227  154  252  136   73  251  129  150

228  216   87  149   81  150  216   39   49   93  141   212   44   212
229   26  168  128   46  222  202   43   33

195  134  136  108  245   14   44   48  231   35   74   244  158    42
73    16   46  242  128  160   39  134   65

116   51  186  131  186   94   43  108  145  145  219   152   62   100
64    59  162  211  108   61  213  164  205

  4  172   79   85  105  161  254  219  161  210   85     7  150   133
238  238   43  234  169  250   49    4  170

210  214  214  110  190  183  112  125   60  172  174   207  129   183
33    16  138   28  172  163  163  214    3

113    5  145   57   68  177   87  208  140  255   13   156  118   145
240   67  159  207  245   58  171  205  143

157  174   94  148  112   21   80  117  238  246   91   179   138   117
179  255  175  232   49  174  197  178  116

202   97  179  194  238  116   93  117   85   15  127    23   241   113
217   54  173   40   28  170   97  118  231

235  212  139  135  174  113  100  115  167   92  111   108    87    22
 53  127  224  31   144   34  113   21   72

188  128  113  163   54   90  151  105  100  140  143    96   102   113
116   74    3  195  248    5  123  210   16

45   181  177   53  214   39   30  230  160   67  157    42    78    93
20   172   79  184    6   67  155   49  122
```

103 109 159 97 160 172 9 1 178 152 29 213 105 77
217 245 199 166 222 29 45 114 251

239 77 203 200 34 178 117 76 101 12 229 214 73 218
143 196 78 193 6 17 0 3 236

234 48 244 174 53 186 222 12 105 146 193 115 100 194
81 170 237 169 132 218 202 79 143

105 49 133 118 155 122 239 177 167 179 202 244 128 243
95 33 173 226 49 46 131 224 166

228 174 225 145 161 142 67 166 214 246 208 37 184 142
222 24 19 69 183 8 54 213 197

14 77 44 203 94 30 41 251 95 192 171 222 78 3
95 3 18 107 64 187 26 85 27

90 138 250 15 147 115 223 141 108 189 51 196 28 152
18 73 3 54 239 137 40 225 0

208 38 69 154 115 183 60 102 152 110 69 113 113 208
51 209 58 9 35 70 104 122 138

2 178 64 12 11 228 165 50 144 162 160 158 119 250
12 252 132 20 133 94 104 143 1

35 96 224 106 6 69 247 160 183 205 137 243 3 56
145 4 117 217 229 3 13 157 115

51 220 188 78 80 65 170 187 130 21 15 163 169 180
31 209 180 87 241 227 241 169 50

50 218 34 223 3 221 222 96 198 242 22 63 185 133
133 159 149 119 85 221 38 157 201

154 151 3 3 98 59 2 2 125 234 69 90 72 238
29 143 130 233 111 65 98 175 158

69 127 228 196 174 206 35 107 47 154 104 48 67 182
197 62 19 58 120 145 79 130 3

50 230 50 248 23 232 209 192 179 64 121 125 181 58
96 210 49 220 38 40 43 182 228

3 210 76 253 9 59 110 203 251 223 232 104 200 115
210 67 97 168 34 152 229 131 194

191 165 169 201 156 61 227 235 206 131 152 118 252 44
11 192 70 228 136 84 82 155 232

Columns 24 through 32

194	57	50	242	148	41	67	248	168
97	232	172	208	225	80	47	124	190
84	1	237	238	191	7	234	209	88
129	150	88	79	97	91	31	164	226
144	138	152	68	185	6	3	78	88
196	167	157	137	41	203	94	169	15
199	80	0	41	244	255	178	91	183
123	59	251	54	50	28	227	240	245
205	106	230	55	198	159	152	124	40
120	76	177	166	157	33	40	23	106
51	172	112	13	41	79	81	172	24
148	240	179	58	7	34	59	131	115
170	87	156	170	73	57	2	56	222
173	144	76	79	248	101	101	185	100
241	30	219	78	243	34	166	17	64
197	43	28	184	58	61	21	246	90
188	71	74	244	245	237	196	53	190
221	142	24	33	174	100	248	41	166
253	124	101	17	14	130	182	163	240
129	243	85	32	153	23	200	0	213
161	59	241	42	100	5	60	85	120
202	122	214	233	55	40	50	70	161
114	134	66	34	46	216	67	11	14
134	202	10	157	19	225	182	24	138
43	49	1	68	1	47	250	104	116
33	232	147	56	201	253	163	209	220
56	236	190	182	4	182	139	222	218
27	3	206	140	224	223	217	5	120
36	196	163	240	90	122	205	186	201
116	242	64	84	184	126	171	217	167
201	208	36	180	247	73	171	186	0
71	236	166	241	39	15	210	244	33

Image Threshold:227.1094

Binarized Feature Matrix:

```
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0
0 1 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 1 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 1 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0
0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 1 0 1 0 1 0 0 0 0 0 1 0 0 0 0 1 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0
1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 1 1 0 0 0
1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1
1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 1 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 0 0 0 1 0 1 0 1 0 0 0 1 0
```

Feature Vector:

```
1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 1 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 0 0 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0
0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 1 0 0 0 1 0 0 0 0 1 0 0 0 1 0 0 1 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 1 0 1 0 1 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1
0 1 0 0 0 0 0 0 0 1 0 0 0 1 0 0 0 0 0 0 0 1 0 0 1 1 0 0 0 0 0 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1
1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 1 0
0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0
0 1 1 0 0 1 0 1 0 0 1 0 0 0 0 1 1 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0
0 0 0 0 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 1 0 0 0 0 0 0 1 0 0 1
0 0 0 0 0 0 1 0 0 0 0 0 0 1 1 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 1 0 0 0 0 0
0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 1 0 0 0 0
0 0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 1 0 0 0 0 0 0 0 0 0
0 0 1 0 0 0 0 0 0 0 0 0 0 0 0 0
```

C. HISTOGRAM

The following figure presents a sample of the MATLAB histogram that is created from the number of elements put into each bin range.

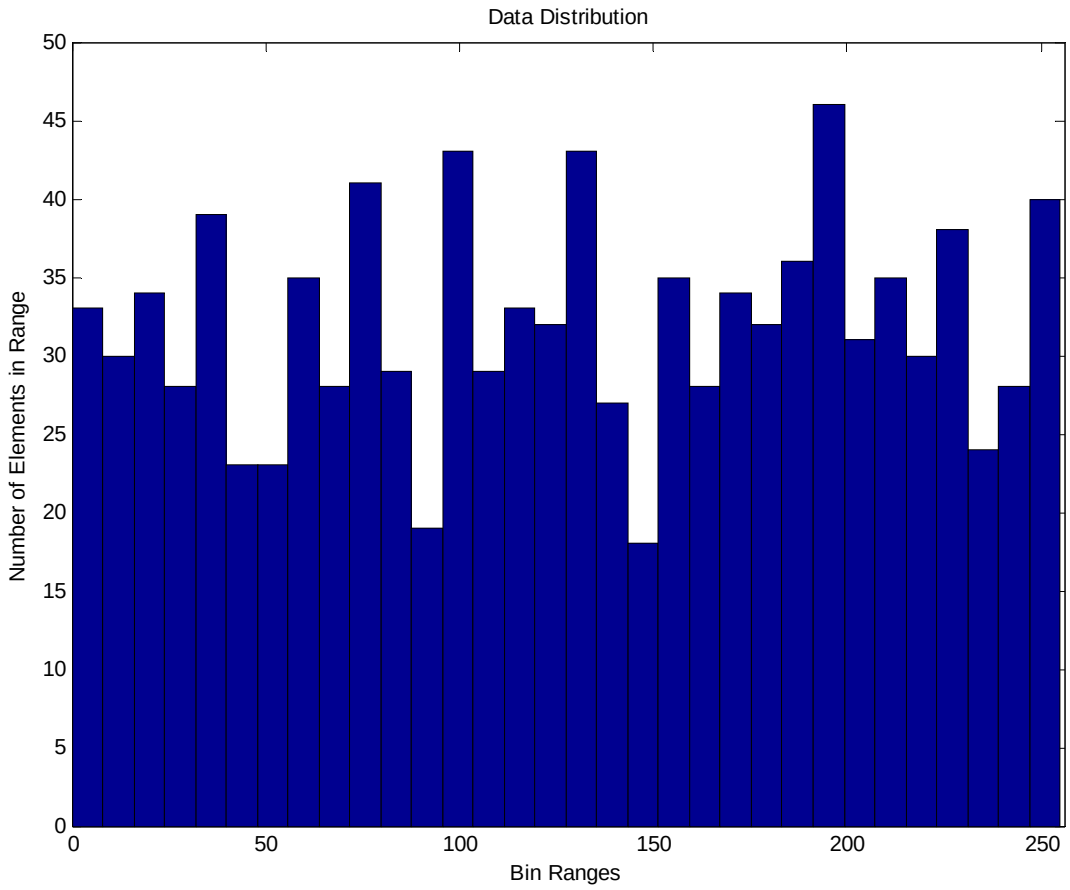

Figure 14. MATLAB Histogram

THIS PAGE INTENTIONALLY LEFT BLANK

APPENDIX B. STANDARD C CODE

A. STANDARD C CODE

```
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <time.h>

int main()
{
int i,j,k;            //j and k are loop variables, i is a temporary
float sum;            //Sum of elements in input matrix
float Cthresh=0.9;    //The CDF threshold
float csum=0.0;       //A cumulative sum of matrix elements
//Total number of elements required by cumulative sum
float cutoff;
unsigned int thresh=0;  //Will be the threshold for binarization step
double layer=5;         //Define QMFB layer to be used
int len;                //length of input array dimensions

//Seed the Random Number Generator
srand((unsigned int) time ((time_t *) NULL));

/*Define input 32 by 32 image matrix of 8 bit values */
unsigned char image[32][32];
/*Define and initialize output feature vector of 32 32-bit values */
unsigned int feature[32]=
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};
/*Define and initialize vector of 32 bins to histogram matrix values */
unsigned int bins[32]=
{0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0};

  len=pow(2,layer);       //Calculate dimensions lengths

  printf("\nQMFB layer: %.0f\n",layer);
  printf("CDF Threshold: %.2f\n",Cthresh);

  printf("\nRandom number input matrix:\n\n");
  /*Assign random integers from 0 to 255 to the 32 by 32 input array
   *This will simulate potential input from the Quadrature Mirror
   *Filter Bank Tree*/
  for(j=0; j < len; j++)
  {
    for(k=0; k < len; k++)
      {
      image[j][k] = (int) (256.0*rand()/(RAND_MAX+1.0));
      //Divide each element by 8 to decide in which bin it belongs
      i = image[j][k] >> 3;
      //Increment count of elements in that bin
      bins[i] += 1;
      printf("%4d",image[j][k]);     //Display array of input numbers
      }
```

```
    printf("\n");
  }

  printf("\nHistogram:\n\n");
  printf("Bin\tElements\tGraph\n");
  for(i=0; i<len; i++)
    {
      printf("%3d\t%4d\t\t",i,bins[i]);
      for(k=0; k < bins[i]; k++)
      printf("*");
      printf("\n");
    }


  i=0;           //i will increment and track which bin is being added
  sum = len*len; //Sum of elements is product of array dimensions
  cutoff = sum*Cthresh;
  printf("\nThe cumulative sum cutoff is: %f\n", cutoff);
  /*Find threshold value by progressively dividing the cumulative sum
   *of bins by total number of elements(1024)*/
  /*Until cutoff value is reached, add number of elements in each
   *successive bin to the cumulative sum */
  while(!(thresh))
    {
      //Add number of elements in this bin to cumulative sum
      csum += bins[i];
      if(csum >= cutoff)             // 0.8*1024(sum)=819
      //Calculate threshold based on where the cutoff point was reached
      thresh = 8*i + 4;
      i++;
    }
  printf("\nCutoff occurs at bin: %d\n ",i-1);
  printf("\nThe threshold is: %d\n",thresh);

  /*Print result of each element greater than threshold test as integer
   * 1 or 0
   *Next to each row of 32, print hexadecimal series of those 32 bits
   *Store each 32-bit row as one 32-bit integer value of vector feature
   *feature contains the variables to pass on to Neural Classification
   *Network*/
  printf("\nBinary Matrix\t\t\t\t\t\t\t Hexadecimal Representation\n");
  for(j=0; j < len; j++)
    {
      for(k=0; k < len; k++)
      {
        printf("%2i",image[j][k]>=thresh);
        //Store result of operation as appropriate bit in vector
        feature[j] += (image[j][k]>=thresh) << 31-k;
      }
      printf("\t%010x\n",feature[j]);
    }

  return 0;
}
```

B. OUTPUT

```
QMFB layer: 5
CDF Threshold: 0.90


Random number input matrix (32x32 output by row):

215 100 200 204 233  50  85 196  71 141 122 160  93 131 243 234 162 183
36  155   4  62  35 205  40 102  33  27 255  55 131 214

156  75 163 134 126 249  74 197 134 197 102 228  72  90 206 235  17 243
134  22  49 169 227  89  16   5 117  16  60 248 230 217

68  138 96 194 131 170 136  10 112 238 238 184  72 189 163 90 176  42
112 225 212 84  58 228  89 175 244 150 168 219 112 236

101 208 175 233 123  55 243 235  37 225 164 110 158  71 201  78 114  57
48   70 142 106  43 232  26  32 126 194 252 239 175  98

191  94  75  59 149  62  39 187  32 203  42 190  19 243  13 133  45  61
204 187 168 247 163 194  23  34 133  20  17  52 118 209

146 193  13  40 255  52 227  32 255  13 222  18   1 236 152  46  41 100
233 209  91 141 148 115 175  25 135 193  77 254 147 224

191 161   9 191 213 236 223 212 250 190 231 251 170 127  41 212 227  19
166  63 161  58 179  81  84  59  18 162  57 166 130 248

71 139 184  28 120 151 241 115  86 217 111   0  88 153 213  59 172 123
123 78 182  46 159  10 105 178 172 163  88  47 155 160

187  84 189  51 235 175 167  65 136  22  66 224 175  23  28  92 147 151
170  73 198  73  84  48 251   0 211  84  48 111 245 235

195 178  31 175  98 198 241 234 220  52 203 140  76 231 232 223 127 147
41   70 221 126 118 217 126  74  46 175 186  35 154 126

214 185  45  56 127  31  35  92  83 238 232 159 214 209 126  85 100 168
155  66  38  18  27 165  93  73  84  23 109 239 149  67

168 195 124  40 226 160 132  53 142 109 212 100  62  83 186 163 252  86
229  34 105   1 200 198  75  29 221 184  12 114 252 181

53  121 221 24 25  98  77 168 207  33  13  13 117 199 177 113  30 150
148 135 152 92 77 227 122  43 156 134 158 152  59 212

17  25 236  43 123  57 211  74  91 224  88 208 168   9  65 199 160 214
78  56  50 156  28 172 200 184  51 102  80 111  59  98

136  39 142   3  97  97  78 188  66 166 141 235 175 207 178  79 165   1
136 216 158 164 132 102  92 184 205 173  39   8  16 175
```

```
48 158 179 145   0   1  78  66 167 219  46  87 170 225 167  80 226  47
40 128 212 172 231  48 100 180 222 140 189 238  59 237

141 238 126 141 240 204 208 152 168 254 239  83 223 150 163 194 198 203
67  154 120  42 203 221 223 170 105 156 152 165 137  37

148  8 179 132 213 131  28 125 130  12 208  98 163 115  36 105  63 104
4  183 146 208 149 114 122 254  15  19 164 152  56  56

161 236 188 118 112 217 243 242 230 196  85 137  56 122 243 119 226 247
47  117 199 196 231  65 194 246  84 103 143 141 159  48

122 92 167 234  53 155 221  28  95  50 165 151 173 152  15 143 144  62
4   88   2 236 153 197 227 238  44 114 124 203 163 247

39  74 225  93 230 191 121  69 242  31 221 159 183 236  47  72  42  51
160 45  32  58 242   3  41  30 118 166 234  25 157  18

100 127 111  75  62 233 144  48   8 110 208 192  91 255   9 134  51 169
179  83 227 165  87  12 196 205 178 174 231  80 192  76

207  48 151  13  25  40  62  34 150  14 227 242  14 236 120  65 150  43
149 121 208 237 134 149 186  57  67 162 137   4 238  88

52 133 102  78 174 165 113  68 180  85  54 194  66 174   4 216 218 153
82 170 134 217  64  65  18 131 227 156 135 210 245 188

88   92  11   6   1 124  74 181 209 129 119  20  48 123 236  11  21  62
182 156  23 246 222  42 121 193 199   1 148 188 190 236

24  201 242  25  70  61 207  24 191  70  44 240 194  24 251 216  87 177
116 111 167  82 153  33  20  96  35 168  29 225 149  53

171 135  79 241 196  31   9 131 102  54 115  41  78 111   1 166  32 118
21  199 201 175 233 222  16  12 134  45 237  28  99 152

163 179 138 104 210 147 235  56 202  95  97  24 206  99 191 239 217 212
182 162 132 159 128 148 171   7 193 153  35  36  50 199

216 188  47 170  80  27 227  26 122  69  51  73 168 242  56 129 199 239
36   75 143 164 223  59 172 161 213 208 197   7 151 158

195 198  72  19 225  44  45  92 113  96 165  25  83 222 155  26 206 191
102  93 100  69 153  17 230 110 225 172 117 120  74  57

62 147  77  32 191 122 124  49 219  34  75  47   0 230  73 207 166 176
44  11 245 198  28 220  53 254 137 170 119 212 228 182

103  49 214  39 172  82  88 136 117 163 183 117 138   1  68  49 177 113
60  167  56  89 131 109  87  12  24 207 225 252 133  72
```

```
Histogram:

Bin   Elements    Graph
  0     24        ************************
  1     27        ***************************
  2     28        ****************************
  3     34        **********************************
  4     30        ******************************
  5     38        ***************************************
  6     35        ***********************************
  7     37        *************************************
  8     27        ***************************
  9     37        *************************************
 10     30        ******************************
 11     29        *****************************
 12     30        ******************************
 13     19        *******************
 14     31        *******************************
 15     36        ************************************
 16     33        *********************************
 17     27        ***************************
 18     31        *******************************
 19     38        ***************************************
 20     46        ***********************************************
 21     44        *********************************************
 22     26        **************************
 23     31        *******************************
 24     38        ***************************************
 25     27        ***************************
 26     34        **********************************
 27     32        ********************************
 28     37        *************************************
 29     43        *******************************************
 30     26        **************************
 31     19        *******************


The cumulative sum cutoff is: 921.599976

Cutoff occurs at bin: 28

The threshold is: 228
```

Binary Matrix	Hexadecimal Representation
00001000000000110000000000001000	0008030008
00000100000100010100000000000110	0004114006
00000000011000000000000100100001	0000600121
00010011000000000000000100001100	001300010c
00000000000001000000010000000000	0000040400
00001000100001000010000000000100	0008842004
00000100101100000000000000000001	0004b00001
00000010000000000000000000000000	0002000000
00001000000000000000000010000011	0008000083
00000011000001100000000000000000	0003060000
00000000011000000000000000000100	0000600004
00000000000000001010000000000010	000000a002
00000000000000000000000000000000	0000000000
00100000000000000000000000000000	0020000000
00000000000100000000000000000000	0000100000
00000000000000000000001000000101	0000000205
01001000011000000000000000000000	0048600000
00000000000000000000000001000000	0000000040
01000011100000100100001001000000	0043824240
00010000000000000000010001000001	0010000441
00001000100001000000001000001000	0008840208
00000100000001000000000000001000	0004040008
00000000000101000000010000000010	0000140402
00000000000000000000000000000010	0000000002
00000000000000100000010000000001	0000020401
00100000000100100000000000000000	0020120000
00010000000000000000001000001000	0010000208
00000010000000010000000000000000	0002010000
00000000000001000100000000000000	0000044000
00000000000000000000000010000000	0000000080
00000000000001000000100001000010	0000040842
00000000000000000000000000000100	0000000004

APPENDIX C. SRC-6 SPECIFIC C CODE

A. SRC-6 SPECIFIC C CODE

1. main.c

```
static char const cvsid[] = "$Id: main.c,v 2.1 2005/06/14 22:16:46 jls
Exp $";

#include <libmap.h>
#include <stdlib.h>
#include <stdio.h>
#include <time.h>

#define SZ 65536

/*Define function prototype for preprocessing subroutine */
void subr (uint64_t* , uint64_t*, int*, uint64_t*, int);

int main ()
{
  //j and k are loop variables, mapnum specifies which MAP to use
  int j,k, mapnum=0;
  //Define new type; 32 element array of 64-bit integers
  typedef uint64_t arr32 [32];
  //Define a pointer to allow an array of type arr32
  arr32 *image;
  uint64_t *feature;   //Pointer to output feature vector
  uint64_t tm;         //Variable to track execution time
  int thold;           //Variable to display calculated threshold value

  //Seed the Random Number Generator
  srand((unsigned int) time ((time_t *) NULL));

  /*Allocate space in memory for a 32 by 32 matrix of 8 byte elements
   *for the input image*/
  image = (arr32*) Cache_Aligned_Allocate(32*32*8);
  //Allocate memory for a 32 element output feature vector
  feature = (uint64_t*) Cache_Aligned_Allocate(32*8);

  printf("\nInput Matrix:\n");
  /*Assign random integers from 0 to 255 to the 32 by 32 input array*/
  for(j=0; j < 32; j++)
  {
    for(k=0; k < 32; k++)
      {
      image[j][k] = (uint64_t) (256.0*rand()/(RAND_MAX+1.0));
      printf("%4d",image[j][k]);     //Display array of input numbers
      }
    printf("\n");
  }

  map_allocate (1);  //Assign a MAP processor to execute the subroutine

  /*Send input image, output vector, time tracker, and MAP specifier to
```

```
   *the subroutine*/
  subr(image, feature, &thold, &tm, mapnum);

  printf("\nThreshold is: %d\n",thold);

  /*Verify proper passing of feature vector from subroutine by printing
   *to screen*/
  printf("\nFeature Vector:\n");
  for(j=0; j < 32; j++)
    printf("%016llx \n",feature[j]);

  //Print subroutine execution time for timing analysis
  printf ("\n%lld clocks\n", tm);

  map_free (1);                       //Free the MAP processor

  return 0;

}
```

2. preproc.mc

```
/* $Id: ex01.mc,v 2.1 2005/06/14 22:16:46 jls Exp $ */

#include <libmap.h>

#define SZ 65536
/*Define the lengths of the rows and columns of the input matrix */
#define len 32

void subr (uint64_t Ain[], uint64_t Din[], int *threshold, uint64_t
*time, int mapnum) {
/*Define OBM Bank A to store values of the input image matrix*/
OBM_BANK_A_2D (A, uint64_t, len,len)
/*Define OBM Bank D to store values of the output feature vector*/
OBM_BANK_D (D, uint64_t, len)
uint64_t t0, t1;         //variables to store timing data
int i,j,k,xs;            //j and k are loop variables, i is a temporary
uint8_t thresh=0;        //Will be the threshold for binarization step
float csum=0.0;          //A cumulative sum of matrix elements
//Total number of elements required by cumulative sum
float cutoff;
float sum;               //Sum of elements in input matrix
float Cthresh=0.9;       //The CDF threshold
/*Define vector of 32 bins to histogram matrix values */
uint16_t bins[len];

    /*Stream input matrix values into OBM Bank A */
    DMA_CPU (CM2OBM, A, MAP_OBM_stripe(1,"A"), Ain, 1,
len*len*sizeof(uint64_t), 0);
    wait_DMA (0);

read_timer (&t0);      /*Take first time reading for timing analysis*/

    /*Clear the bin array and Bank D so they can receive data */
    for(j=0; j < len; j++)
```

```
     {
      bins[j]=0;
      D[j]=0;
     }

    for(xs=0; xs < len*len; xs++)
     {
   //Perform loop flattening to reduce execution time
         cg_count_ceil_32(1, 0, xs==0, len-1, &k);
         cg_count_ceil_32(k==0, 0, xs==0, SZ, &j);
/*Divide each element in input matrix by 8 to decide in which bin it
 *belongs*/
         i = A[j][k] >> 3;
         //Increment count of elements in that bin
         bins[i] += 1;
     }

/*Find threshold value by progressively dividing the cumulative sum of
 *bins by total number of elements(1024)*/
//i will increment and track which bin is being added to cumulative sum
i=0;
//Sum of elements is product of array dimensions
sum = len*len;
cutoff = sum*Cthresh;
/*Find threshold value by progressively dividing the cumulative sum of
 *bins by total number of elements(1024)*/
/*Until cutoff value is reached, add number of elements in each
 *successive bin to the cumulative sum */
  while(!(thresh))
    {
//Add number of elements in this bin to cumulative sum
      csum += bins[i];
//Calculate threshold based on where the cutoff point was reached
      if(csum >= cutoff)
            thresh = 8*i + 4;
      i++;
    }

  /* Calculate result of each element greater than threshold test as
   *integer 1 or 0   */
  /* Store each 32-bit row as one sign extended 64-bit integer value of
   *Bank D        */
  /* OBM Bank D contains the variables to pass on to the Neural
   *Classification Network*/
    for(xs=0; xs < len*len; xs++)
     {
   //Perform loop flattening to reduce execution time
         cg_count_ceil_32(1, 0, xs==0, len-1, &k);
         cg_count_ceil_32(k==0, 0, xs==0, SZ, &j);
         //Store result of comparison as appropriate bit in vector
         D[j] += (A[j][k]>=thresh) << 31-k;
     }

/*Take second time reading for timing analysis*/
    read_timer (&t1);
```

```
/*Subtract times to determine execution duration*/
    *time = t1 - t0;

/*Store the treshold value to be passed back for display*/
    *threshold = thresh;

/* Stream values from OBM Bank D to variable to be passed back to main
 *program */
    DMA_CPU (OBM2CM, D, MAP_OBM_stripe(1,"D"), Din, 1,
len*sizeof(uint64_t), 0);
    wait_DMA (0);
    }
```

3. Make File

```
# ----------------------------------
# User defines FILES, MAPFILES, and BIN here
# ----------------------------------
FILES             = main.c

MAPFILES    = preproc.mc

BIN         = preproc

# ----------------------------------
# Multi chip info provided here
# (Leave commented out if not used)
# ----------------------------------
#PRIMARY    = <primary file 1>   <primary file 2>

#SECONDARY  = <secondary file 1> <secondary file 2>

#CHIP2            = <file to compile to user chip 2>

#-----------------------------------
# User defined directory of code routines
# that are to be inlined
#-----------------------------------

#INLINEDIR  =

# ----------------------------------
# User defined macros info supplied here
#
# (Leave commented out if not used)
# ----------------------------------
#MACROS           = <directory-name/macro-file>

#MY_BLKBOX  = <directory-name/blackbox-file>
#MY_NGO_DIR       = <directory-name>
#MY_INFO    = <directory-name/info-file>
# ----------------------------------
# Floating point macros selection
# ----------------------------------

#FPMODE           = SRC_IEEE_V1 # Default SRC version IEEE
```

```
#FPMODE           = SRC_IEEE_V2 # Size reduced SRC IEEE with
                        # special rounding mode
# ----------------------------------
# User supplied MCC and MFTN flags
# ----------------------------------

MY_MCCFLAGS       = -log
MY_MFTNFLAGS      = -log

# ----------------------------------
# User supplied flags for C & Fortran compilers
# ----------------------------------

CC          = icc       # icc   for Intel cc for Gnu
FC          = ifort     # ifort for Intel f77 for Gnu
#LD         = ifort     # for Fortran or C/Fortran mixed
LD          = icc # for C codes

MY_CFLAGS   =
MY_FFLAGS   =
MY_LDFLAGS  =     # Flags to include libs if needed
# ----------------------------------
# VCS simulation settings
# (Set as needed, otherwise just leave commented out)
# ----------------------------------

#USEVCS           = yes # YES or yes to use vcs instead of vcsi
#VCSDUMP    = yes # YES or yes to generate vcd+ trace dump
# ----------------------------------
# No modifications are required below
# ----------------------------------
MAKIN   ?= $(MC_ROOT)/opt/srcci/comp/lib/AppRules.make
include $(MAKIN)
```

B. OUTPUT

```
Input Matrix:
215 100 200 204 233  50  85 196  71 141 122 160  93 131 243 234 162 183
36  155   4  62  35 205  40 102  33  27 255  55 131 214

156  75 163 134 126 249  74 197 134 197 102 228  72  90 206 235  17 243
134  22  49 169 227  89  16   5 117  16  60 248 230 217

68  138  96 194 131 170 136  10 112 238 238 184  72 189 163  90 176  42
112 225 212  84  58 228  89 175 244 150 168 219 112 236

101 208 175 233 123  55 243 235  37 225 164 110 158  71 201  78 114  57
48   70 142 106  43 232  26  32 126 194 252 239 175  98

191  94  75  59 149  62  39 187  32 203  42 190  19 243  13 133  45  61
204 187 168 247 163 194  23  34 133  20  17  52 118 209

146 193  13  40 255  52 227  32 255  13 222  18   1 236 152  46  41 100
233 209  91 141 148 115 175  25 135 193  77 254 147 224
```

```
191 161   9 191 213 236 223 212 250 190 231 251 170 127  41 212 227  19
166  63 161  58 179  81  84  59  18 162  57 166 130 248

71  139 184  28 120 151 241 115  86 217 111   0  88 153 213  59 172 123
123  78 182  46 159  10 105 178 172 163  88  47 155 160

187  84 189  51 235 175 167  65 136  22  66 224 175  23  28  92 147 151
170  73 198  73  84  48 251   0 211  84  48 111 245 235

195 178  31 175  98 198 241 234 220  52 203 140  76 231 232 223 127 147
41  70  221 126 118 217 126  74  46 175 186  35 154 126

214 185  45  56 127  31  35  92  83 238 232 159 214 209 126  85 100 168
155  66  38  18  27 165  93  73  84  23 109 239 149  67

168 195 124  40 226 160 132  53 142 109 212 100  62  83 186 163 252  86
229  34 105   1 200 198  75  29 221 184  12 114 252 181

53  121 221  24  25  98  77 168 207  33  13  13 117 199 177 113  30 150
148 135 152  92  77 227 122  43 156 134 158 152  59 212

17  25 236  43 123  57 211  74  91 224  88 208 168   9  65 199 160 214
78  56  50 156  28 172 200 184  51 102  80 111  59  98

136  39 142   3  97  97  78 188  66 166 141 235 175 207 178  79 165   1
136 216 158 164 132 102  92 184 205 173  39   8  16 175

48 158 179 145   0   1  78  66 167 219  46  87 170 225 167  80 226  47
40 128 212 172 231  48 100 180 222 140 189 238  59 237

141 238 126 141 240 204 208 152 168 254 239  83 223 150 163 194 198 203
67  154 120  42 203 221 223 170 105 156 152 165 137  37

148   8 179 132 213 131  28 125 130  12 208  98 163 115  36 105  63 104
4   183 146 208 149 114 122 254  15  19 164 152  56  56

161 236 188 118 112 217 243 242 230 196  85 137  56 122 243 119 226 247
47  117 199 196 231  65 194 246  84 103 143 141 159  48

122 92 167 234  53 155 221  28  95  50 165 151 173 152  15 143 144  62
4   88   2 236 153 197 227 238  44 114 124 203 163 247

39  74 225  93 230 191 121  69 242  31 221 159 183 236  47  72  42  51
160 45  32  58 242   3  41  30 118 166 234  25 157  18

100 127 111  75  62 233 144  48   8 110 208 192  91 255   9 134  51 169
179  83 227 165  87  12 196 205 178 174 231  80 192  76

207  48 151  13  25  40  62  34 150  14 227 242  14 236 120  65 150  43
149 121 208 237 134 149 186  57  67 162 137   4 238  88

52 133 102  78 174 165 113  68 180  85  54 194  66 174   4 216 218 153
82 170 134 217  64  65  18 131 227 156 135 210 245 188

88   92  11   6   1 124  74 181 209 129 119  20  48 123 236  11  21  62
182 156  23 246 222  42 121 193 199   1 148 188 190 236
```

```
24  201 242  25  70  61 207  24 191  70  44 240 194  24 251 216  87 177
116 111 167  82 153  33  20  96  35 168  29 225 149  53

171 135  79 241 196  31   9 131 102  54 115  41  78 111   1 166  32 118
21  199 201 175 233 222  16  12 134  45 237  28  99 152

163 179 138 104 210 147 235  56 202  95  97  24 206  99 191 239 217 212
182 162 132 159 128 148 171   7 193 153  35  36  50 199

216 188  47 170  80  27 227  26 122  69  51  73 168 242  56 129 199 239
36  75  143 164 223  59 172 161 213 208 197   7 151 158

195 198  72  19 225  44  45  92 113  96 165  25  83 222 155  26 206 191
102  93 100  69 153  17 230 110 225 172 117 120  74  57

62 147  77  32 191 122 124  49 219  34  75  47   0 230  73 207 166 176
44  11 245 198  28 220  53 254 137 170 119 212 228 182

103  49 214  39 172  82  88 136 117 163 183 117 138   1  68  49 177 113
60  167  56  89 131 109  87  12  24 207 225 252 133  72

Threshold is: 228


Feature Vector (in Hexadecimal):
0000000008030008
0000000004114006
0000000000600121
000000001300010c
0000000000040400
0000000008842004
0000000004b00001
0000000002000000
0000000008000083
0000000003060000
0000000000600004
000000000000a002
0000000000000000
0000000020000000
0000000000100000
0000000000000205
0000000048600000
0000000000000040
0000000043824240
0000000010000441
0000000008840208
0000000004040008
0000000000140402
0000000000000002
0000000000020401
0000000020120000
0000000010000208
0000000002010000
0000000000044000
0000000000000080
0000000000040842
0000000000000004
```

```
Time:
16821 clocks
```

LIST OF REFERENCES

[1] David Caliga and David Peter Barker, "*Delivering Acceleration: The Potential for Increased HPC Application Performance Using Reconfigurable Logic*," ACM 1-58113-293-X/01/0011, November 2001.

[2] Information, Signal, Images et ViSion. "Time-Frequncy Toolbox – Tutorial." 26 October 2005, http://gdr-isis.org/tftb/tutorial/node7.html. Last Accessed: 18 May 2005.

[3] SRC Computers, Inc., "SRC Carte Training Course," presented by David Caliga at a private training session, Colorado Springs, Colorado, November 2005.

[4] "SRC-6 MAP© Hardware Guide," SRC-005-05, SRC Computers, Inc., Colorado Springs, Colorado, May 26, 2004.

[5] "SRC-6 C Programming Environment V2.1 Guide," SRC-007-16, SRC Computers Inc., Colorado Springs, Colorado, August 31, 2005.

[6] P. E. Pace, *Detecting and Classifying Low Probability Of Intercept Radar,* Artech House Inc., Boston, Massachusetts, 2004.

[7] Sony Akkarakaran and P. P. Vaidyanathan, "Bifrequency and bispectrum maps: a new look at multirate systems with stochastic inputs," *IEEE Transactions On Signal Processing*, Vol. 48, No. 3, pp. 723-736, March 2000.

[8] E. Zilberman and P. E. Pace, "Autonomous Cropping and Feature Extraction Using Time-Frequency Marginal Distributions for LPI Radar Classification," Eighth IASTED International Conference on Signal and Image Processing, Honolulu, Hawaii, August 14 - 16, 2006.

[9] D. Moloney. The Code Project. "C++/Mex wrapper adds microsecond resolution timer to Matlab under WinXP." 25 August, 2005. http://www.codeproject.com/cpp/Matlab_Microsecond_Timer.asp. Last Accessed: 19 June 2006.

THIS PAGE INTENTIONALLY LEFT BLANK

INITIAL DISTRIBUTION LIST

1. Defense Technical Information Center
   Ft. Belvoir, Virginia

2. Dudley Knox Library
   Naval Postgraduate School
   Monterey, California

3. Chairman, Code EC
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

4. Alan Hunsberger
   National Security Agency
   Ft. Meade, Maryland

5. Douglas J. Fouts
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

6. Herschel H. Loomis
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

7. Phillip E. Pace
   Department of Electrical and Computer Engineering
   Naval Postgraduate School
   Monterey, California

8. David Caliga
   SRC Computers, Inc.
   Colorado Springs, Colorado

9. Jon Huppenthal
   SRC Computers, Inc.
   Colorado Springs, Colorado